

Dal catalogo Apogeo Education

Informatica

- Bolchini, Brandolesi, Salice, Sciuto, *Reti logiche*, seconda edizione
Bruni, Corradini, Gervasi, *Programmazione in Java*, seconda edizione
Cabodi, Camurati, Pasini, Patti, Vendraminetto, *Dal problema al programma. Introduzione al problem solving in linguaggio C*
Cabodi, Camurati, Pasini, Patti, Vendraminetto, *Ricorsione e problem-solving. Strategie algoritmiche in linguaggio C*
Collins, *Algoritmi e strutture dati in Java*
Coppola, Mizzaro, *Laboratorio di programmazione in Java*
Deitel, Deitel, *C++. Fondamenti di programmazione*
Deitel, Deitel, *C++. Tecniche avanzate di programmazione*
Della Mea, Di Gaspero, Scagnetto, *Programmazione web lato server*, seconda edizione aggiornata
Di Noia, De Virgilio, Di Sciascio, Donnini, *Semantic web*
Facchinetti, Larizza, Rubini, *Programmare in C. Concetti di base e tecniche avanzate*
Hanly, Koffman, *Problem solving e programmazione in C*
Hennessy, Patterson, *Architettura degli elaboratori*
Horstmann, *Concetti di informatica e fondamenti di Java*, quinta edizione
Horstmann, *Concetti di informatica e fondamenti di Python*
King, *Programmazione in C*
Laganà, Righi, Romani, *Informatica. Concetti e sperimentazioni*, seconda edizione
Lambert, *Programmazione in Python*
Lombardo, Valle, *Audio e multimedia*
Malik, *Programmazione in C++*
Mazzanti, Milanese, *Programmazione di applicazioni grafi che in Java*
Peterson, Davie, *Reti di calcolatori*, terza edizione
Schneider, Gersting, *Informatica*
Tarabella, *Musica informatica*

Algoritmi e strutture dati in Java

Michael T. Goodrich

Roberto Tamassia

Michael H. Goldwasser

Edizione italiana a cura di
Marcello Dalpasso



Algoritmi e strutture dati in Java

Autori: Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Titolo originale: Data Structures & Algorithms in Java 6th edition

Copyright © 2014 by John Wiley & Sons, Inc. All rights reserved.

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Traduzione e revisione: Marcello Dalpasso

Impaginazione elettronica: Grafica editoriale

© Copyright 2015 by Maggioli S.p.A.

Maggioli Editore è un marchio di Maggioli S.p.A.

Azienda con sistema qualità certificato ISO 9001: 2008

47822 Santarcangelo di Romagna (RN) • Via del Carpino, 8

Tel. 0541/628111 • Fax 0541/622595

www.maggiolieditore.it

e-mail: clienti.editore@maggioli.it

**Diritti di traduzione, di memorizzazione elettronica, di riproduzione
e di adattamento, totale o parziale con qualsiasi mezzo sono riservati per tutti i Paesi.**

Finito di stampare nel mese di settembre 2015

nello stabilimento Maggioli S.p.A.

Santarcangelo di Romagna (RN)

A Karen, Paul, Anna e Jack
Michael T. Goodrich

A Isabel
Roberto Tamassia

A Susan, Calista e Maya
Michael H. Goldwasser

Sommario

Presentazione della edizione italiana	xv
Prefazione	xvii
Capitolo 1 – Introduzione a Java	1
1.1 Per cominciare.....	1
1.1.1 I tipi fondamentali.....	3
1.2 Classi e oggetti	4
1.2.1 Creare e usare oggetti	5
1.2.2 Definire una classe.....	8
1.3 Stringhe, involucri, array ed enumerazioni.....	16
1.4 Espressioni.....	22
1.4.1 Letterali.....	22
1.4.2 Operatori.....	23
1.4.3 Conversioni di tipo	27
1.5 Controllo di flusso	29
1.5.1 Gli enunciati if e switch.....	29
1.5.2 Cicli.....	31
1.5.3 Enunciati per il controllo di flusso esplicito.....	35
1.6 Casi semplici di input/output	36
1.7 Un esempio di programma	39
1.8 Pacchetti e importazione	42
1.9 Sviluppo del software.....	44
1.9.1 Progettazione	44
1.9.2 Pseudocodice	46
1.9.3 Scrittura del codice	47

1.9.4 Documentazione e stile	48
1.9.5 Collaudo e debugging	51
1.10 Esercizi.....	53
Capitolo 2 – Progettazione orientata agli oggetti.....	57
2.1 Obiettivi, principi e schemi ricorrenti	57
2.1.1 Obiettivi della progettazione orientata agli oggetti	57
2.1.2 Principi della progettazione orientata agli oggetti.....	58
2.1.3 Schemi di progetto (<i>design pattern</i>).....	60
2.2 Ereditarietà.....	61
2.2.1 Estensione della classe CreditCard	62
2.2.2 Polimorfismo e smistamento dinamico.....	64
2.2.3 Gerarchie di ereditarietà	66
2.3 Interfacce e classi astratte	72
2.3.1 Interfacce in Java	72
2.3.2 Ereditarietà multipla per interfacce	74
2.3.3 Classi astratte	75
2.4 Eccezioni	77
2.4.1 Catturare eccezioni	77
2.4.2 Lanciare eccezioni	80
2.4.3 La gerarchia delle eccezioni in Java	81
2.5 Cast e tipi generici	83
2.5.1 Cast	83
2.5.2 Programmazione mediante tipi generici.....	86
2.6 Classi annidate	91
2.7 Esercizi.....	92
Capitolo 3 – Strutture dati fondamentali	97
3.1 Array	97
3.1.1 Memorizzare in un array i punteggi di un gioco	97
3.1.2 Ordinare un array	103
3.1.3 I metodi di java.util per gli array e i numeri casuali	105
3.1.4 Una semplice crittografia che usa array di caratteri.....	108
3.1.5 Array bidimensionali e giochi posizionali.....	111
3.2 Liste semplicemente concatenate	115
3.2.1 Realizzare una lista semplicemente concatenata	119
3.3 Liste concatenate circolari.....	121
3.3.1 Pianificazione circolare (<i>round robin</i>).....	121
3.3.2 Progettare e realizzare una lista concatenata circolare.....	122
3.4 Liste doppiamente concatenate	125
3.4.1 Realizzare una lista doppiamente concatenata	127
3.5 Verifica di equivalenza	130
3.5.1 Verifica di equivalenza tra array	131
3.5.2 Verifica di equivalenza tra liste concatenate	132
3.6 Clonare strutture dati.....	133
3.6.1 Clonare array	134
3.6.2 Clonare liste concatenate	136
3.7 Esercizi.....	137

Capitolo 4 – Analisi di algoritmi.....	141
4.1 Analisi sperimentali	142
4.1.1 Superare le analisi sperimentali	145
4.2 Le sette funzioni usate in questo libro	147
4.2.1 Confrontare velocità di crescita.....	154
4.3 Analisi asintotica.....	155
4.3.1 La notazione “O-grande”	156
4.3.2 Analisi comparativa	160
4.3.3 Esempi di analisi di algoritmi.....	162
4.4 Semplici tecniche di dimostrazione	169
4.4.1 Dimostrare con un esempio.....	169
4.4.2 Dimostrare per contrapposizione o contraddizione	169
4.4.3 Dimostrare per induzione o mediante invariante di ciclo	170
4.5 Esercizi.....	173
Capitolo 5 – Ricorsione.....	181
5.1 Esempi di ricorsione	182
5.1.1 Funzione fattoriale	182
5.1.2 Disegnare un righello in pollici.....	184
5.1.3 Ricerca binaria.....	187
5.1.4 File system	189
5.2 Analisi di algoritmi ricorsivi.....	193
5.3 Ulteriori esempi di ricorsione.....	197
5.3.1 Ricorsione lineare	197
5.3.2 Ricorsione binaria o doppia	201
5.3.3 Ricorsione multipla.....	202
5.4 Progettazione di algoritmi ricorsivi	204
5.5 Ricorsioni fuori controllo.....	205
5.5.1 Massima profondità di ricorsione in Java	208
5.6 Eliminare la ricorsione in coda	209
5.7 Esercizi.....	211
Capitolo 6 – Pile, code e code doppie	215
6.1 Pile (<i>stack</i>).....	215
6.1.1 La pila come tipo di dato astratto	216
6.1.2 Una semplice implementazione di pila basata su array	219
6.1.3 Realizzare una pila con una lista semplicemente concatenata.....	222
6.1.4 Invertire un array usando una pila.....	223
6.1.5 Corrispondenza tra parentesi e tra marcatori HTML	224
6.2 Code (<i>queue</i>).....	227
6.2.1 Il tipo di dato astratto “coda”	228
6.2.2 Implementazione di coda basata su array	230
6.2.3 Coda realizzata con una lista semplicemente concatenata	234
6.2.4 Coda circolare	235
6.3 Code doppie	236
6.3.1 Il tipo di dato astratto “coda doppia”.....	237
6.3.2 Implementazione di una coda doppia	238
6.3.3 Code doppie nel Java Collections Framework	239
6.4 Esercizi.....	240

Capitolo 7 – Liste e iteratori.....	247
7.1 Liste	247
7.2 Liste con indice	249
7.2.1 Array dinamici	252
7.2.2 Implementare un array dinamico	253
7.2.3 Analisi ammortizzata degli array dinamici	254
7.2.4 La classe <code>StringBuilder</code> di Java	259
7.3 Liste posizionali	259
7.3.1 Posizioni	261
7.3.2 Il tipo di dato astratto “lista posizionale”	262
7.3.3 Implementazione con lista doppiamente concatenata	265
7.4 Iteratori.....	270
7.4.1 L’interfaccia <code>Iterable</code> e il ciclo <code>for-each</code> in Java.....	271
7.4.2 Implementazione di iteratori	272
7.5 L’infrastruttura Java Collections Framework	276
7.5.1 Iteratori di lista in Java.....	277
7.5.2 Confronto con il nostro ADT “lista posizionale”	278
7.5.3 Algoritmi basati su liste nel Java Collections Framework	279
7.6 Ordinare una lista posizionale	280
7.7 Caso di studio: gestire frequenze di accesso	281
7.7.1 Implementazione con una lista ordinata	282
7.7.2 Uso di una lista con l’euristica <code>move-to-front</code>	284
7.8 Esercizi.....	287
Capitolo 8 – Alberi.....	295
8.1 Alberi generici	295
8.1.1 Alberi: definizioni e proprietà	296
8.1.2 L’albero come tipo di dato astratto	299
8.1.3 Calcolare profondità e altezza	301
8.2 Alberi binari.....	304
8.2.1 L’albero binario come tipo di dato astratto	306
8.2.2 Proprietà degli alberi binari	308
8.3 Implementare alberi.....	310
8.3.1 Una struttura concatenata per alberi binari	310
8.3.2 Albero binario rappresentato con un array	316
8.3.3 Struttura concatenata per alberi generici	318
8.4 Algoritmi di attraversamento di alberi	319
8.4.1 Attraversamenti in pre-ordine e post-ordine per alberi generici.....	320
8.4.2 Attraversamento in ampiezza (<i>breadth-first</i>) di un albero	321
8.4.3 Attraversamento in ordine simmetrico di un albero binario.....	322
8.4.4 Implementare attraversamenti di alberi in Java.....	324
8.4.5 Applicazioni di attraversamenti di alberi.....	328
8.4.6 Percorso di Eulero	333
8.5 Esercizi.....	335
Capitolo 9 – Code prioritarie	345
9.1 La coda prioritaria come tipo di dato astratto	345
9.1.1 Priorità	345

9.1.2 La coda prioritaria come ADT	346
9.2 Implementare una coda prioritaria	347
9.2.1 L'oggetto composito Entry	347
9.2.2 Confrontare chiavi totalmente ordinate.....	348
9.2.3 La classe di base AbstractPriorityQueue	350
9.2.4 Realizzare una coda prioritaria con una lista non ordinata	351
9.2.5 Realizzare una coda prioritaria con una lista ordinata	353
9.3 Heap	354
9.3.1 La struttura dati <i>heap</i>	355
9.3.2 Implementare una coda prioritaria con uno heap.....	356
9.3.3 Analisi di una coda prioritaria realizzata con uno heap.....	365
9.3.4 Costruzione di uno heap dal basso verso l'alto (<i>bottom-up</i>)*	366
9.3.5 Utilizzo della classe java.util.PriorityQueue	370
9.4 Ordinare con una coda prioritaria.....	371
9.4.1 Ordinamento per selezione e ordinamento per inserimento	372
9.4.2 Heap Sort	374
9.5 Code prioritarie flessibili	375
9.5.1 Entry consapevoli della propria posizione	377
9.5.2 Implementare una coda prioritaria flessibile	378
9.6 Esercizi.....	381
 Capitolo 10 – Mappe, tabelle hash e skip list	387
10.1 Mappe	387
10.1.1 La mappa come tipo di dato astratto	388
10.1.2 Applicazione: frequenza delle parole in un testo.....	390
10.1.3 La classe di base AbstractMap	391
10.1.4 Una semplice implementazione di mappa non ordinata	393
10.2 Tabelle hash.....	394
10.2.1 Funzioni di hash	395
10.2.2 Schemi di gestione delle collisioni	401
10.2.3 Fattore di carico, rehashing ed efficienza.....	404
10.2.4 Implementazione di tabella hash in Java	406
10.3 Mappe ordinate	412
10.3.1 Tabelle di ricerca ordinate	413
10.3.2 Due applicazioni di mappe ordinate	417
10.4 Skip list	420
10.4.1 Operazioni di ricerca e modifica in una skip list*	422
10.4.2 Analisi probabilistica della prestazioni di una skip list	426
10.5 Insiemi, multi-insiemi e multi-mappe	429
10.5.1 L'insieme come tipo di dato astratto	429
10.5.2 Il multi-insieme come tipo di dato astratto	431
10.5.3 Il tipo di dato astratto multi-mappa	432
10.6 Esercizi.....	434
 Capitolo 11 – Alberi di ricerca	443
11.1 Alberi di ricerca binari.....	443
11.1.1 Ricerca in un albero di ricerca binario	444
11.1.2 Inserimenti e rimozioni	446

11.1.3	Implementazione in Java	449
11.1.4	Prestazioni di un albero di ricerca binario.....	453
11.2	Alberi di ricerca bilanciati.....	454
11.2.1	Infrastruttura Java per bilanciare alberi di ricerca.....	456
11.3	Alberi AVL.....	461
11.3.1	Operazioni di modifica	463
11.3.2	Implementazione in Java	468
11.4	Alberi <i>splay</i>	469
11.4.1	<i>Splaying</i> o estensione	470
11.4.2	Quando si esegue l'estensione?.....	471
11.4.3	Implementazione in Java	474
11.4.4	Analisi ammortizzata dell'estensione o <i>splaying</i> *	476
11.5	Alberi (2, 4).....	481
11.5.1	Alberi di ricerca a più vie.....	481
11.5.2	Operazioni in un albero (2, 4)	484
11.6	Alberi rosso-nero.....	491
11.6.1	Operazioni in un albero rosso-nero	493
11.6.2	Implementazione in Java	502
11.7	Esercizi.....	505
Capitolo 12 – Ordinamento e selezione		513
12.1	Ordinamento per fusione (<i>merge-sort</i>)	513
12.1.1	Dividi-e-conquista (<i>divide-and-conquer</i>).....	513
12.1.2	Implementazione di merge-sort basata su array	517
12.1.3	Il tempo d'esecuzione di merge-sort	519
12.1.4	Merge-sort ed equazioni di ricorrenza*	521
12.1.5	Implementazioni alternative di merge-sort	522
12.2	Ordinamento <i>quick-sort</i>	525
12.2.1	Quick-sort con scelta casuale del pivot	531
12.2.2	Ulteriori ottimizzazioni per quick-sort.....	533
12.3	Approfondimento algoritmico per lo studio dell'ordinamento	536
12.3.1	Limite inferiore asintotico per l'ordinamento	536
12.3.2	Ordinamento in tempo lineare: <i>bucket-sort</i> e <i>radix-sort</i>	538
12.4	Confronto tra algoritmi di ordinamento.....	541
12.5	Selezione	543
12.5.1	<i>Prune-and-Search</i> (riduzione-e-ricerca)	543
12.5.2	Quick-select probabilistico	544
12.5.3	Analsi del quick-select probabilistico	545
12.6	Esercizi.....	546
Capitolo 13 – Elaborazione di testi.....		555
13.1	Abbondanza di testi digitalizzati	555
13.1.1	Notazioni per stringhe di caratteri	556
13.2	Algoritmi di <i>pattern matching</i>	557
13.2.1	Forza bruta	558
13.2.2	L'algoritmo di Boyer-Moore	559
13.2.3	L'algoritmo di Knuth-Morris-Pratt	563

13.3	Trie	567
13.3.1	Trie standard	568
13.3.2	Trie compresso	571
13.3.3	Trie dei suffissi	572
13.3.4	Indicizzazione nei motori di ricerca	575
13.4	Compressione del testo e metodo <i>greedy</i>	576
13.4.1	L'algoritmo di codifica di Huffman	577
13.4.2	Il metodo <i>greedy</i>	578
13.5	Programmazione dinamica	579
13.5.1	Moltiplicazione matriciale a catena	579
13.5.2	DNA e allineamento di sequenze di caratteri	582
13.6	Esercizi	585
Capitolo 14 – Algoritmi per grafi.....		593
14.1	Grafi	593
14.1.1	Il grafo come tipo di dato astratto	599
14.2	Strutture dati per grafi	600
14.2.1	La struttura a lista di lati (<i>edge list</i>)	600
14.2.2	La struttura a lista di adiacenze (<i>adjacency list</i>)	603
14.2.3	La struttura a mappa di adiacenze (<i>adjacency map</i>)	605
14.2.4	La struttura a matrice di adiacenze (<i>adjacency matrix</i>)	606
14.2.5	Implementazione in Java	607
14.3	Attraversamenti di un grafo	610
14.3.1	Attraversamento in profondità (<i>depth-first search</i>)	611
14.3.2	Implementazione di DFS e sue estensioni	616
14.3.3	Attraversamento in ampiezza (<i>breadth-first search</i>)	620
14.4	Chiusura transitiva	623
14.5	Grafi orientati aciclici	627
14.5.1	Ordinamento topologico	628
14.6	Ricerca dei percorsi più brevi	631
14.6.1	Grafi pesati	631
14.6.2	L'algoritmo di Dijkstra	633
14.7	Alberi ricoprenti minimi	644
14.7.1	L'algoritmo di Prim-Jarník	645
14.7.2	L'algoritmo di Kruskal	648
14.7.3	Partizioni disgiunte e strutture <i>union-find</i>	658
14.8	Esercizi	663
Capitolo 15 – Gestione della memoria e B-alberi		675
15.1	Gestione della memoria	675
15.1.1	Strutture di tipo <i>stack</i> nella Java Virtual Machine	676
15.1.2	Riservare spazio nella memoria <i>heap</i>	678
15.1.3	Garbage collection	680
15.2	Gerarchie di memoria e <i>caching</i>	682
15.2.1	Sistemi di memoria	682
15.2.2	Strategie di gestione della memoria <i>cache</i>	683

15.3 Ricerche esterne e <i>B</i> -alberi	688
15.3.1 Alberi (<i>a, b</i>)	689
15.3.2 <i>B</i> -alberi	691
15.4 Ordinamento nella memoria esterna	692
15.4.1 Fusione a più vie	693
15.5 Esercizi	695
Appendice – Proprietà matematiche utili.....	699
Bibliografia.....	707
Indice analitico	713

Presentazione della edizione italiana

L'indiscusso successo di questo testo, che negli Stati Uniti è giunto ormai alla sesta edizione (la prima è addirittura del 1998) ed è affiancato da analoghe versioni in linguaggio C++ e Python, mi ha spinto a considerare con grande favore l'idea di curare la sua traduzione italiana. L'insieme degli argomenti trattati dagli Autori rispecchia in modo molto fedele il contenuto dell'insegnamento di "Dati e Algoritmi" o "Fondamenti di Informatica 2" che, nelle sue varie denominazioni, caratterizza oggi in modo piuttosto omogeneo i corsi di laurea del settore dell'informazione delle università italiane.

In questa versione, con l'apporto del nuovo autore, Michael Goldwasser, il classico testo di Goodrich e Tamassia, pur continuando a presentare in modo sistematico e matematicamente rigoroso gli aspetti teorici della materia, dedica una maggiore attenzione al *Java Collections Framework*, che fa parte della libreria dell'ambiente di sviluppo del linguaggio Java, Standard Edition, e contiene la maggior parte delle strutture dati e degli algoritmi elementari e di livello intermedio: un'evoluzione che sarà sicuramente apprezzata da quei docenti che, come me, utilizzano il testo da molti anni.

La scelta del linguaggio Java come ausilio di programmazione per la presentazione degli argomenti e la realizzazione dei progetti è da me pienamente condivisa, dal momento che inseguo informatica da molti anni usando questo linguaggio, con piena soddisfazione. In particolare, in questa edizione gli Autori hanno aggiornato il loro codice sorgente in modo da sfruttare al meglio le nuove caratteristiche di Java 7 (e 8): per portare un esempio, l'ampio utilizzo dell'inferenza dei tipi rende molto più snella la creazione di esemplari di classi che usino la programmazione per tipi generici. Questo adeguamento del codice rende ovviamente necessario l'utilizzo della versione 7 o superiore, anche se non è difficile modificare il sorgente per adeguarlo, in caso di necessità, alla versione 5 o 6.

Il testo è, inoltre, correddato da un'ampia e variegata raccolta di esercizi, suddivisi in tre tipologie: "riepilogo e approfondimento", "creatività" e "progettazione". I primi solleciti-

Computer Science del Department of Mathematics and Computer Science della Saint Louis University, mentre in precedenza è stato docente del Department of Computer Science della Loyola University Chicago. La sua ricerca ha come oggetto principalmente la progettazione e la realizzazione di algoritmi e le sue pubblicazioni riguardano prevalentemente agli algoritmi approssimati, il calcolo online, la biologia computazionale e la geometria computazionale. È anche attivo nella comunità dell'insegnamento dell'informatica.

Ringraziamenti

Sono talmente numerose le persone che ci hanno aiutato nello sviluppo di questo libro, negli ultimi dieci anni, che è veramente difficile citarle tutte. Vogliamo, però, cogliere l'occasione per rinnovare i nostri ringraziamenti ai molti collaboratori alla ricerca e agli assistenti didattici che ci hanno fornito elementi utili per dar forma alle precedenti versioni di questo libro: i loro suggerimenti hanno avuto importanza vitale anche per questa edizione.

Per questa sesta edizione, in particolare, siamo in debito con i revisori esterni e con i lettori, per i loro molti commenti e le loro critiche costruttive. Tra loro, vogliamo in particolare ringraziare: Sameer O.Abuafarreh (North Dakota State University), Mary Boelk (Marquette University), Frederick Crabbe (United States Naval Academy), Scot Drysdale (Dartmouth College), David Eisner, Henry A. Edinger (Rochester Institute of Technology), Chun-Hsi Huang (University of Connecticut), John Lasseter (Hobart and William Smith Colleges), Yupeng Lin, Suely Oliveira (University of Iowa), Vincent van Oostrom (Utrecht University), Justus Piater (University of Innsbruck), Victor I. Shtern (Boston University), Tim Soethout e molti altri revisori anonimi.

Molti amici e colleghi hanno fornito suggerimenti che ci hanno portato a migliorare il testo e siamo particolarmente grati a Erin Chambers, Karen Goodrich, David Letscher, David Mount e Ioannis Tollis per i loro commenti veramente approfonditi. Inoltre, ringraziamo in particolar modo David Mount per il suo contributo alla trattazione della ricorsione e per le molte figure con cui l'ha arricchita.

Abbiamo molto apprezzato la straordinaria squadra di Wiley, e in particolare Beth Lang Golub, per l'entusiasmo con il quale ci ha aiutato in questo progetto, dall'inizio alla fine, nonché le persone del Product Solutions Group, Mary O'Sullivan e Ellen Keohane, che ci hanno consentito di portarlo a termine. La qualità di questo libro è stata significativamente migliorata grazie all'attento lavoro di Julie Kennedy, che ha revisionato i testi con grande cura dei dettagli, e gli ultimi mesi della produzione sono stati curati da Joyce Poh.

Infine, vogliamo ringraziare con calore Karen Goodrich, Isabel Cruz, Susan Goldwasser, Giuseppe Di Battista, Franco Preparata, Ioannis Tollis e i nostri genitori per i consigli, l'incoraggiamento e il supporto che ci hanno fornito nelle varie fasi di questo lavoro, oltre a Calista e Maya Goldwasser per averci consigliato con competenza artistica in merito a molte delle illustrazioni del libro. Il ringraziamento più importante, però, va a tutte quelle persone che ci hanno sempre ricordato che, nella vita, esistono cose che vanno al di là della scrittura di libri.

*Michael T. Goodrich
Roberto Tamassia
Michael H. Goldwasser*

1

Introduzione a Java

1.1 Per cominciare

Per progettare strutture dati e algoritmi è necessario comunicare al computer istruzioni dettagliate e un modo eccellente per farlo consiste nell'utilizzare un linguaggio di programmazione di alto livello, come Java. In questo capitolo presentiamo una panoramica del linguaggio di programmazione Java, per poi proseguire la discussione nel capitolo successivo, dove ci concentreremo sui principi della progettazione orientata agli oggetti. Ipotizziamo, qui, che i lettori sappiano programmare in un linguaggio di alto livello, anche se non necessariamente in Java. Questo libro non fornisce una descrizione completa del linguaggio Java (a tale scopo esistono numerosi manuali di riferimento), bensì ne presenta quegli aspetti che saranno usati nei capitoli successivi.

Iniziamo, quindi, questa introduzione a Java con un programma che visualizza sullo schermo il messaggio "Hello Universe!": la Figura 1.1 lo presenta evidenziando in grande dettaglio le sue componenti.

In Java, gli enunciati eseguibili devono essere scritti all'interno di funzioni, chiamate *metodi*, che appartengono alla definizione di una *classe*. La classe `Universe`, nel nostro primo esempio, è veramente molto semplice: ha un unico metodo statico di nome `main` ("principale"), che è anche il primo metodo eseguito quando si mette in esecuzione un programma Java. Un insieme di enunciati racchiuso tra una coppia di parentesi graffe corrispondenti (una aperta, {" e una chiusa, "}) definisce un *blocco* (di codice) all'interno del programma. Si noti come l'intera definizione della classe `Universe` sia delimitata da tali parentesi, così come il corpo del metodo principale.

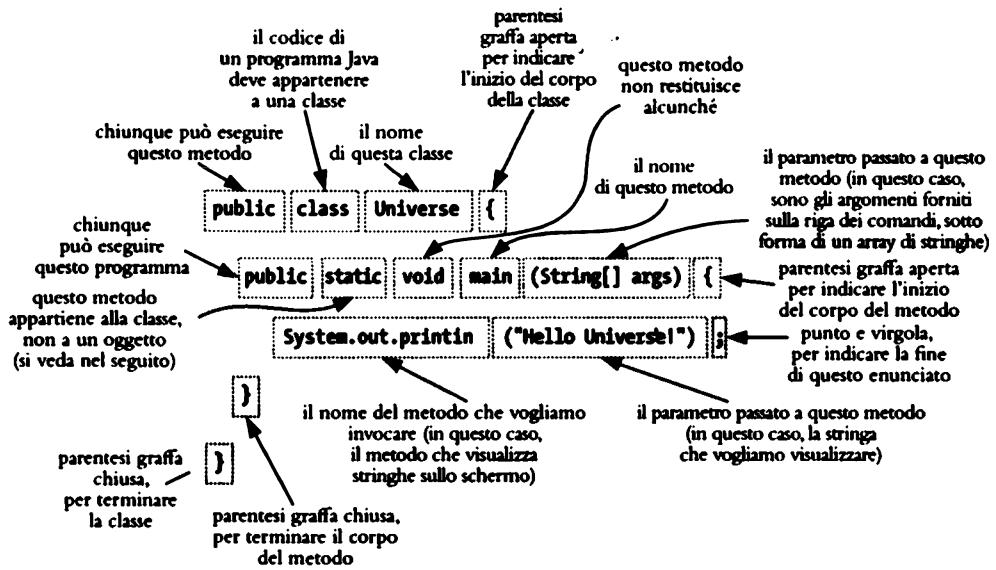


Figura 1.1: Un programma che visualizza il messaggio "Hello Universe!".

Il nome di una classe, di un metodo o di una variabile, in Java, viene detto **identificatore** e può essere una stringa di caratteri qualsiasi che inizi con una lettera dell'alfabeto e sia composta soltanto da lettere, cifre numeriche e caratteri di sottolineatura (*underscore*): le "lettere" e le "cifre numeriche" possono appartenere a qualunque lingua scritta definita nell'insieme dei caratteri Unicode. La Tabella 1.1 elenca le uniche eccezioni a questa regola generale per definire gli identificatori in Java.

Tabella 1.1: Elenco delle parole riservate del linguaggio Java: non possono essere usate come identificatori di classe, metodo o variabile.

Parole riservate				
abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

Commenti

Oltre a enunciati eseguibili e dichiarazioni, Java consente ai programmatori di inserire commenti nel codice: si tratta, come in tutti i linguaggi di programmazione, di annotazioni fornite per lettori umani e non vengono elaborate dal compilatore. Java consente due for-

me di commenti: i commenti “su una sola riga” (*inline comment*) e i commenti “a blocco” (*block comment*). Per iniziare un commento che si estende su una sola riga si usa, in Java, una coppia di barre oblique, “//”: i caratteri seguenti sulla riga saranno ignorati dal compilatore fino al termine della riga stessa. Ad esempio:

```
// Commento che si estende fino al termine della riga.
```

Java consente anche di scrivere commenti che si estendono su più righe, sotto forma di commento “a blocco”: lo si inizia con la sequenza di caratteri “/*” e lo si termina con “*/”, come in questo esempio

```
/*
 * Questo è un commento "a blocco".
 */
```

I commenti a blocco che iniziano con la sequenza “/**” (si noti la presenza del secondo asterisco) hanno un significato speciale e consentono al programma Javadoc di leggerli e di usarli per generare in modo automatico la documentazione del software: ne parleremo nel Paragrafo 1.9.4.

1.1.1 I tipi fondamentali

Per i tipi di dati più diffusi, Java mette a disposizione i seguenti *tipi fondamentali* (detti anche *tipi primitivi*):

boolean	un valore booleano: true (vero) o false (falso)
char	un carattere a 16 bit nello standard Unicode
byte	un numero intero con segno rappresentato in complemento a due con 8 bit
short	un numero intero con segno rappresentato in complemento a due con 16 bit
int	un numero intero con segno rappresentato in complemento a due con 32 bit
long	un numero intero con segno rappresentato in complemento a due con 64 bit
float	un numero in virgola mobile a 32 bit nello standard IEEE 754-1985
double	un numero in virgola mobile a 64 bit nello standard IEEE 754-1985

Una variabile di uno di questi tipi memorizza semplicemente un valore di tale tipo. Le costanti numeriche intere, come 14 o 195, sono di tipo **int**, a meno che non siano seguite da una “L” (maiuscola o minuscola), nel qual caso sono di tipo **long**. Le costanti numeriche in virgola mobile, come 3.1416 o 6.022e23, sono di tipo **double**, a meno che non siano seguite da una “F” (maiuscola o minuscola), nel qual caso sono di tipo **float**. Il (frammento di) Codice 1.1 mostra la dichiarazione, e in alcuni casi l'inizializzazione (o “assegnazione del valore iniziale”), di variabili dei vari tipi fondamentali.

Codice 1.1: Dichiarazione e inizializzazione di alcune variabili dei vari tipi fondamentali.

```
1 boolean flag = true;
2 boolean verbose, debug;           // due variabili dichiarate, ma non inizializzate
3 char grade = 'A';
4 byte b = 12;
5 short s = 24;
```

```

6 int i, j, k = 257;           // tre variabili dichiarate, solo k inizializzata
7 long l = 890L;              // si noti qui l'uso di "L"
8 float pi = 3.1416F;         // si noti qui l'uso di "F"
9 double e = 2.71828, a = 6.022e23; // entrambe le variabili vengono inizializzate

```

Si noti come sia possibile dichiarare (e inizializzare) più variabili dello stesso tipo con un unico enunciato, come nelle righe 2, 6 e 9 di questo esempio. In questo frammento di codice, le variabili `verbose`, `debug`, `i` e `j` rimangono non inizializzate. Le variabili dichiarate localmente all'interno di un blocco di codice devono essere inizializzate prima del loro primo utilizzo.

Una caratteristica comoda di Java riguarda la dichiarazione di variabili di tipi fondamentali come variabili di esemplare di una classe (si veda il prossimo paragrafo): se non vengono inizializzate esplicitamente, lo sono a un valore predefinito. In particolare, le variabili di un tipo numerico sono inizializzate a zero, le variabili booleane sono inizializzate a `false` e le variabili di tipo carattere sono inizializzate al carattere nullo.

1.2 Classi e oggetti

Nei programmi Java più complessi, gli “attori” principali sono gli *oggetti*. Ogni oggetto è un *esemplare* (o *istanza*) di una classe, la quale svolge il ruolo di *tipo* dell’oggetto e di suo schema progettuale, definendo i dati memorizzati all’interno dell’oggetto e i metodi per accedere a tali dati e modificarli. I *membri* chiave di una classe, in Java, sono i seguenti:

- *Variabili di esemplare* o di istanza, dette anche *campi*, che rappresentano i dati associati a un certo oggetto. Le variabili di esemplare devono avere un *tipo*, che può essere uno dei tipi fondamentali (come `int`, `float` o `double`) oppure una qualsiasi classe (e in quest’ultimo caso si parla anche di *tipo riferimento*, per i motivi che spiegheremo a breve).
- *Metodi*, che in Java sono blocchi di codice che vengono invocati (o “chiamati”) per eseguire azioni, in analogia con quanto avviene con funzioni e procedure in altri linguaggi di alto livello. I metodi possono ricevere parametri sotto forma di argomenti dell’invocazione e il loro comportamento può dipendere sia dall’oggetto con il quale vengono invocati sia dai valori dei parametri che vengono passati. Un metodo che restituisce un’informazione all’invocante senza modificare alcuna variabile di esemplare viene anche detto *metodo di accesso*, mentre un *metodo modificatore*, quando viene invocato, può modificare alcune delle variabili di esemplare.

A titolo di esempio, il Codice 1.2 riporta la definizione completa di una classe molto semplice, chiamata `Counter` (cioè “contatore”), alla quale faremo ripetutamente riferimento nel seguito di questo paragrafo.

Codice 1.2: La classe `Counter` definisce un semplice contatore, che può essere interrogato, incrementato e riportato al valore iniziale.

```

1 public class Counter {
2     private int count;          // una semplice variabile di esemplare, numerica intera
3     public Counter() { }        // costruttore di default (assegna 0 a count)
4     public Counter(int initial) { count = initial; } // costruttore alternativo

```

```

5   public int getCount() { return count; }           // un metodo di accesso
6   public void increment() { count++; }             // un metodo modificatore
7   public void increment(int delta) { count += delta; } // un metodo modificatore
8   public void reset() { count = 0; }                // un metodo modificatore
9 }
```

Questa classe definisce un'unica variabile di esemplare, `count`, che viene dichiarata alla riga 2 e rappresenta il valore di conteggio del contatore. Come già detto, tale variabile avrà zero come valore predefinito, a meno che non venga inizializzata in altro modo.

La classe contiene due metodi speciali, che prendono il nome di *costruttori* (definiti alle righe 3 e 4), oltre a un metodo di accesso (alla riga 5) e a tre metodi modificatori (righe 6, 7 e 8). Diversamente dalla prima classe, `Universe`, vista nel paragrafo precedente, la classe `Counter` non ha un metodo `main`, per cui non può essere eseguita come se fosse un programma completo. Infatti, lo scopo della classe `Counter` è quello di creare esemplari che verranno usati all'interno di un programma più grande.

1.2.1 Creare e usare oggetti

Prima di analizzare i dettagli sintattici che caratterizzano la definizione della classe `Counter`, preferiamo descrivere come si creano e si utilizzano gli esemplari di `Counter`. A questo scopo, il Codice 1.3 presenta una nuova classe, `CounterDemo`.

Codice 1.3: Un esempio di utilizzo di esemplari della classe `Counter`.

```

1 public class CounterDemo {
2   public static void main(String[] args) {
3     Counter c;          // dichiara una variabile; nessun contatore ancora costruito
4     c = new Counter();  // costruisce un contatore e ne assegna il riferimento a c
5     c.increment();       // ne incrementa il valore di un'unità
6     c.increment(3);    // ne incrementa ulteriormente il valore di tre unità
7     int temp = c.getCount(); // temp assumerà il valore 4
8     c.reset();          // azzerà il contatore
9     Counter d = new Counter(5); // dichiara e costruisce un contatore con valore 5
10    d.increment();      // il valore del secondo contatore diventa 6
11    Counter e = d;      // assegna a e il riferimento allo stesso oggetto d
12    temp = e.getCount(); // varrà 6 (e e d si riferiscono allo stesso contatore)
13    e.increment(2);    // il valore di e (e anche di d) diventa 8
14  }
15 }
```

In Java esiste una distinzione molto importante tra la gestione delle variabili di tipi fondamentali e quella delle variabili il cui tipo è una classe. Nella riga 3 dell'esempio precedente viene dichiarata una nuova variabile, `c`, usando questa sintassi:

```
Counter c;
```

Questo stabilisce che l'identificatore `c` rappresenta una variabile di tipo `Counter`, ma non crea un esemplare di `Counter`. In Java, le classi sono *tipi riferimento* e una variabile di tale tipo (come `c` in questo esempio) è una *variabile riferimento*. Una variabile riferimento è in grado di memorizzare la posizione (cioè l'*indirizzo in memoria*) di un oggetto, creato come

esemplare di una determinata classe: possiamo assegnarle un valore che faccia riferimento a un oggetto esistente o a un oggetto nuovo, appena costruito. Una variabile riferimento può anche contenere un valore speciale, `null`, che rappresenta la mancanza di un oggetto effettivo cui fare riferimento.

In Java, si crea un nuovo oggetto usando l'operatore `new` seguito dall'invocazione di un costruttore della classe di cui si vuole creare un esemplare (un costruttore è un metodo che ha sempre lo stesso nome della classe in cui è definito). L'operatore `new` restituisce un *riferimento* all'esemplare appena creato e tale riferimento viene solitamente assegnato a una variabile (*riferimento*), per poterlo usare in seguito.

Alla riga 4 del Codice 1.3 si costruisce un nuovo oggetto di tipo `Counter`, il cui riferimento viene assegnato alla variabile `c`: questo processo sfrutta una delle forme del costruttore, `Counter()`, che non vuole argomenti tra le parentesi (tale costruttore privo di parametri viene detto *costruttore di default* o predefinito). Alla riga 9 viene costruito un altro contatore, questa volta usando una diversa forma di costruttore che riceve un parametro, consentendo di specificare durante l'invocazione un valore iniziale diverso da zero per il contatore.

Durante la creazione di un nuovo esemplare di una classe si possono distinguere tre fasi:

- Viene posizionato nella memoria dinamica un nuovo oggetto e tutte le sue variabili di esemplare vengono inizializzate ai relativi valori predefiniti, che sono `null` per le variabili riferimento e zero per tutti i tipi fondamentali, tranne per le variabili di tipo `boolean` che vengono inizializzate al valore `false`.
- Viene invocato il costruttore del nuovo oggetto, usando i parametri specificati. Il costruttore può, quindi, assegnare ad alcune variabili di esemplare un valore più significativo, eseguendo anche tutte le ulteriori elaborazioni che si rendano necessarie in seguito alla creazione dell'oggetto.
- Dopo che il costruttore ha terminato il proprio compito, l'operatore `new` restituisce un riferimento all'oggetto appena creato (cioè il suo indirizzo in memoria). Se l'espressione ha la forma di un enunciato di assegnazione, tale indirizzo viene memorizzato in una *variabile oggetto*, in modo che questa *si riferisca* all'oggetto appena creato.

L'operatore "punto"

Una variabile che fa riferimento a un oggetto viene principalmente utilizzata per accedere ai membri definiti nella classe di cui tale oggetto è un esemplare, cioè per accedere ai metodi e alle variabili di esemplare associati all'oggetto stesso. Questo accesso avviene mediante l'operatore "punto". Per invocare un metodo associato a un oggetto si usa il nome di una variabile che faccia riferimento a tale oggetto, seguito dall'operatore punto e dal nome del metodo e dai suoi parametri. Ad esempio nel Codice 1.3, abbiamo invocato `c.increment()` alla riga 5, `c.increment(3)` alla riga 6, `c.getCount()` alla riga 7 e `c.reset()` alla riga 8. Se l'operatore viene usato con un riferimento che ha, in quel momento, il valore `null`, l'ambiente di esecuzione di Java (*Java runtime environment*) lancia un'eccezione di tipo `NullPointerException`.

Se all'interno di una classe sono definiti più metodi aventi lo stesso nome, il sistema di esecuzione di Java usa il metodo che corrisponde sia all'effettivo numero di parametri indicati come argomenti sia ai loro rispettivi tipi. Ad esempio, la nostra classe `Counter` dispone di due metodi di nome `increment`: una forma che non vuole parametri e un'altra che necessita di un parametro. L'ambiente Java determina quale sia la versione da invocare

nel momento in cui valuta ciascuna delle invocazioni, come `c.increment()` e `c.increment(3)`. L'insieme del nome di un metodo, dei suoi parametri e dei relativi tipi, prende il nome di **firma (signature)** del metodo, nel senso che per individuare quale versione del metodo sia effettivamente necessaria per portare a termine una sua determinata invocazione servono proprio quelle informazioni. Si noti, però, che la firma di un metodo non comprende il tipo del valore (eventualmente) restituito dal metodo stesso, per cui Java non consente la definizione di due metodi aventi la stessa firma, nemmeno se restituiscano valori di tipi diversi.

Una variabile riferimento v può essere vista come un “puntatore” (*pointer*) a un oggetto o , come se tale variabile contenesse un telecomando con il quale si può controllare remotamente l'oggetto (cioè il dispositivo telecomandato). La variabile è, quindi, uno strumento mediante il quale si individua un oggetto e gli si chiede di compiere delle azioni o di fornire l'accesso ai propri dati: un concetto illustrato dalla Figura 1.2. Proseguendo con l'analogia, un riferimento `null` è un contenitore per un telecomando, che però al momento è vuoto.

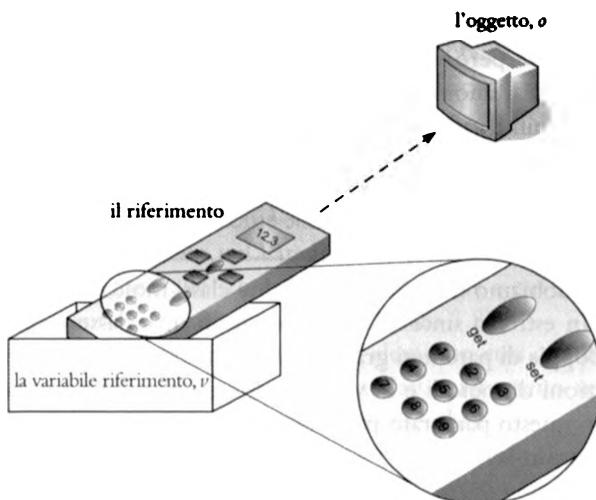


Figura 1.2: Raffigurazione della relazione esistente tra oggetti e variabili riferimento: quando assegniamo a una variabile il riferimento a un oggetto (cioè il suo indirizzo in memoria), è come memorizzassimo in essa un telecomando con il quale si può pilotare l'oggetto stesso.

In effetti, possono anche esistere più variabili che fanno riferimento al medesimo oggetto: ciascuna di esse può essere utilizzata per invocare metodi associati a tale oggetto. Una situazione di questo tipo corrisponderebbe al fatto di poter disporre di più telecomandi per uno stesso dispositivo, ciascuno dei quali possa essere utilizzato per modificare lo stato del dispositivo (ad esempio, cambiare il canale su cui è sintonizzato un televisore). Evidentemente, se viene usato un determinato telecomando per modificare lo stato del dispositivo che può pilotare, tale modifica al dispositivo (che è unico) verrà vista anche tramite gli altri telecomandi; allo stesso modo, se si usa una certa variabile riferimento per modificare lo stato di un oggetto, allora tale stato modificato è visibile anche tramite qualunque altra variabile che faccia riferimento allo stesso oggetto. Questo comportamento è dovuto al fatto che,

nonostante esistano più riferimenti, l'oggetto a cui essi puntano è unico, lo stesso per tutti.

Tornando al nostro esempio `CounterDemo`, l'esemplare di contatore costruito alla riga 9 in questo modo:

```
Counter d = new Counter(5);
```

è diverso dall'esemplare il cui riferimento è stato precedentemente assegnato alla variabile `c`. Al contrario, il comando visibile nella riga 11:

```
Counter e = d;
```

non ha come risultato la costruzione di un nuovo esemplare di `Counter`, bensì dichiara una nuova *variabile riferimento* di nome `e` le assegna un riferimento all'esemplare, già esistente, a cui punta la variabile `d`. A quel punto, le due variabili, `d` ed `e`, identificano il medesimo oggetto, per cui l'invocazione `d.getCount()` si comporta esattamente come si comporterebbe `e.getCount()`. Analogamente, l'invocazione del metodo di aggiornamento `e.increment(2)` agisce sullo stesso oggetto identificato da `d`.

È forse utile osservare, però, che l'individuazione di uno stesso oggetto da parte di due diverse variabili non è un fenomeno permanente: in qualsiasi momento è possibile assegnare a una delle variabili il riferimento a un nuovo esemplare o a un diverso esemplare già esistente, oppure anche `null`.

1.2.2 Definire una classe

Fino a questo punto abbiamo definito soltanto due classi molto semplici: la classe `Universe` e la classe `Counter`. In estrema sintesi, la definizione di una classe è un blocco di codice, delimitato da una coppia di parentesi graffe, una aperta e una chiusa, all'interno delle quali si trovano dichiarazioni di metodi e di variabili di esemplare, che costituiscono i membri della classe stessa. In questo paragrafo prenderemo in esame in modo più approfondito la definizione delle classi in Java.

Modificatori

Subito prima della definizione di una classe, di una variabile di esemplare o di un metodo, in Java, si possono inserire delle parole chiave, dette *modificatori*, che hanno il compito di rappresentare un vincolo aggiuntivo rispetto a tale definizione.

Modificatori per il controllo dell'accesso

Il primo insieme di modificatori di cui parliamo riguarda i *modificatori per il controllo dell'accesso*, che, appunto, hanno il compito di regolare il *livello di accesso* (detto anche *visibilità*) che la classe che si sta definendo concede alle altre classi, nel contesto di un programma Java che sia costituito da più classi. Il fatto che una classe possa concedere un accesso limitato ai propri membri è un principio basilare della progettazione orientata agli oggetti e prende il nome di *incapsulamento* (si veda il Paragrafo 2.1). Ecco un elenco dei modificatori per il controllo dell'accesso e il loro significato:

- Il modificatore `public` indica che qualsiasi classe può accedere all'elemento così qualificato. Ad esempio, nel Codice 1.2 abbiamo scritto:

```
public class Counter {
```

e, di conseguenza, qualunque altra classe (come `CounterDemo`) può costruire nuovi esemplari della classe `Counter`, così come dichiarare variabili e parametri di tipo `Counter`. In Java, ogni classe pubblica deve essere definita in un diverso file sorgente, che si deve chiamare `nomeClasse.java`, dove `nomeClasse` è, appunto, il nome della classe (ad esempio, la definizione della classe `Counter` deve trovarsi nel file `Counter.java`).

Il fatto che a un *metodo* di una classe sia assegnato l'accesso `public` consente a qualsiasi altra classe di invocare tale metodo. Ad esempio, il fatto che alla riga 5 del Codice 1.2 abbiamo scritto:

```
public int getCount() { return count; }
```

consente, in particolare, alla classe `CounterDemo` di invocare `c.getCount()`.

Se una variabile di esemplare è stata dichiarata pubblica, si può usare la notazione "punto" per accedervi direttamente dal codice di qualunque altra classe che sia in possesso di un riferimento a un esemplare della classe in cui è definita la variabile. Ad esempio, se la variabile `count` di `Counter` fosse stata dichiarata pubblica (ma non è così), allora la classe `CounterDemo` avrebbe potuto ispezionare o modificare tale variabile usando una sintassi come `c.count`.

- Il modificatore `protected` indica che l'accesso all'elemento così qualificato è consentito soltanto ai seguenti gruppi di altre classi:
 - classi che sono state definite come *sottoclassi* della classe in oggetto, mediante l'ereditarietà (di cui parleremo nel Paragrafo 2.2);
 - classi che appartengono allo stesso *pacchetto (package)* a cui appartiene la classe in oggetto (argomento che tratteremo nel Paragrafo 1.8).
 - Il modificatore `private` indica che l'accesso all'elemento così qualificato è consentito soltanto al codice appartenente alla medesima classe: nemmeno una sottoclasse, così come nessun'altra classe, può avere accesso a tali membri.
- Ad esempio, la variabile di esemplare `count` è stata definita nella classe `Counter` con livello di accesso privato, quindi possiamo leggere il suo valore o modificarlo dall'interno di qualsiasi metodo di tale classe (come `getCount`, `increment` e `reset`), ma altre classi, come `CounterDemo`, non possono accedere direttamente a quel campo. Ovviamente, è possibile (e l'abbiamo fatto) definire metodi pubblici che consentano a tali classi esterne di ottenere comportamenti che dipendano dal valore della variabile `count`.
- Infine, evidenziamo il fatto che, se non viene specificato in modo esplicito nessun modificatore di controllo dell'accesso, l'elemento in oggetto è caratterizzato da un livello di accesso che viene detto *privato di pacchetto (package-private)*: questo consente l'accesso all'elemento da parte di qualsiasi altra classe appartenente allo stesso pacchetto (Paragrafo 1.8), ma ciò non è consentito a classi o sottoclassi in pacchetti diversi.

Il modificatore `static`

In Java, il modificatore `static` può essere associato alla dichiarazione di qualsiasi variabile o metodo di una classe (o anche di una classe annidata, *nested class*, di cui parleremo nel Paragrafo 2.6).

Quando una variabile di una classe viene dichiarata `static`, il suo valore risulta essere associato alla classe nella sua interezza, piuttosto che a ciascun suo singolo esemplare. Le

variabili statiche sono, quindi, utilizzate per memorizzare informazioni "globali" relative a una classe (ad esempio, si potrebbe usare una variabile statica per tenere traccia del numero complessivo di esemplari di una classe che sono stati creati). Le variabili statiche di una classe esistono, all'interno di un programma in esecuzione, anche se non è stato creato nessun suo esemplare.

Quando, invece, è un metodo di una classe a essere dichiarato **static**, anch'esso viene associato alla classe e non a un suo particolare esemplare. Questo significa che il metodo non va invocato usando uno specifico esemplare della classe con la normale notazione "punto", ma lo si invoca solitamente usando il nome della classe.

Come esempio, nel pacchetto `java.lang` (che fa parte della distribuzione standard di Java) è presente la classe `Math` che contiene molti metodi statici, tra i quali il metodo `sqrt` che calcola la radice quadrata di un numero. Per calcolare una radice quadrata, non c'è bisogno di creare un esemplare della classe `Math`: il metodo va invocato usando una sintassi come `Math.sqrt(2)`, dove il nome della classe, `Math`, viene usato come qualificatore del metodo prima dell'operatore "punto".

I metodi statici possono essere utili per mettere a disposizione comportamenti, relativi a una classe, che non abbiano bisogno di esaminare o modificare lo stato di un particolare esemplare della classe.

Il modificatore `abstract`

Un metodo di una classe può essere dichiarato **abstract**, nel qual caso ne va specificata la firma ma non se ne descrive l'implementazione nel corpo. I metodi astratti sono una caratteristica avanzata della programmazione orientata agli oggetti che va di pari passo con l'ereditarietà e verrà trattata nel Paragrafo 2.3.3. In breve, qualunque sottoclasse di una classe che abbia metodi astratti deve fornire un'implementazione concreta di ciascuno di essi.

Una classe che contenga almeno un metodo astratto deve essere anch'essa dichiarata esplicitamente **abstract**, perché, in pratica, è incompleta (ed è possibile dichiarare astratta una classe anche se non contiene alcun metodo astratto). Coerentemente, Java non consente la costruzione di esemplari di una classe astratta, anche se si possono dichiarare variabili riferimento di un tipo astratto.

Il modificatore `final`

Una variabile che venga dichiarata con il modificatore **final** ("definitiva") può essere inizializzata al momento della sua dichiarazione, ma non le si può poi assegnare un nuovo valore. Se si tratta di una variabile di un tipo fondamentale, essa diventa a tutti gli effetti una costante. Se è una variabile riferimento a essere **final**, allora questa farà sempre riferimento allo stesso oggetto, anche se quest'ultimo potrà modificare il proprio stato interno. Se una variabile di una classe viene dichiarata **final**, questa verrà solitamente dichiarata anche **static**, perché sarebbe veramente uno spreco di risorse disporre di variabili di esemplare che memorizzino tutte lo stesso valore, dal momento che tale valore può essere sicuramente condiviso dall'intera classe.

Indicare che un metodo o un'intera classe è **final** ha una conseguenza completamente diversa, significativa soltanto nell'ambito dell'ereditarietà: un metodo **final** non può essere sovrascritto in una sottoclasse, mentre addirittura non è possibile definire una sottoclasse di una classe **final**.

Dichiarare variabili di esemplare

Durante la definizione di una classe, si dichiarano tutte le sue variabili di esemplare, in numero qualsiasi. Un principio fondamentale della progettazione orientata agli oggetti prevede che ciascun esemplare di una classe gestisca il proprio insieme di variabili di esemplare (ed è proprio per questo motivo che tali variabili vengono dette *di esemplare*). Quindi, nel caso della classe Counter, ogni suo esemplare memorizzerà il proprio valore di count, indipendente da quello di altri esemplari.

Ecco, nel seguito, la sintassi richiesta per dichiarare variabili di esemplare in una classe (le parentesi quadre racchiudono parti facoltative della dichiarazione):

```
[modificatori] tipo identificatore, [ = valoreIniziale1 ], identificatore, [ = valoreIniziale2 ],
```

Nel caso della classe Counter, avevamo dichiarato questa variabile:

```
private int count;
```

dove **private** è il modificatore, **int** è il tipo e **count** è l'identificatore. Dato che non abbiamo indicato un valore iniziale per la variabile, le verrà assegnato automaticamente il valore predefinito previsto per il tipo fondamentale “numero intero”, che è zero.

Dichiarare metodi

La definizione di un metodo è costituita da due parti: la *firma (signature)*, che definisce il nome e i parametri del metodo, e il *corpo (body)*, che definisce le azioni svolte dal metodo. La firma del metodo specifica come questo debba essere invocato, mentre il suo corpo dice cosa avverrà all'oggetto con cui il metodo viene invocato. Ecco la sintassi per la definizione di un metodo:

```
[modificatori] tipoDelValoreRestituito nomeDelMetodo(tipo1 param1, ..., tipon paramn) {  
    // corpo del metodo...  
}
```

Ogni parte di questa dichiarazione ha un compito specifico. Abbiamo già parlato del significato dei *modificatori* come **public**, **private** e **static**. Il *tipoDelValoreRestituito* definisce il tipo del valore che viene restituito dal metodo. Il *nomeDelMetodo* può essere qualunque identificatore valido in Java. L'elenco dei parametri e dei loro tipi dichiara le variabili locali che corrispondono ai valori che vengono passati al metodo come argomenti durante la sua invocazione. Ogni dichiarazione del tipo di un parametro, *tipo_i*, può essere il nome di qualunque tipo di dato in Java, mentre ciascun *param_i* deve essere un diverso identificatore valido in Java. L'elenco dei parametri può anche essere vuoto: in tal caso il metodo va invocato senza passare alcun valore. Queste variabili parametro, così come le variabili di esemplare della classe, possono essere utilizzate nel codice del corpo del metodo, dove si possono anche invocare altri metodi della stessa classe.

Quando si invoca un metodo (non statico) di una classe, lo si fa mediante un esemplare di tale classe, il cui stato può essere modificato dal metodo stesso. Ad esempio, il seguente metodo della classe Counter modifica della quantità specificata il valore del contatore:

```
public void increment(int delta) {
    count += delta;
}
```

Come si può notare, il corpo di questo metodo usa `count`, che è una variabile di esemplare, e `delta`, che è un parametro.

Tipo del valore restituito

La definizione di un metodo deve specificare il tipo del valore che verrà restituito dal metodo al termine della propria esecuzione. Se un metodo non restituisce un valore (come il metodo `increment` della classe `Counter`), si deve usare la parola chiave `void`. In Java, per restituire un valore si deve usare, all'interno del corpo del metodo, la parola chiave `return`, seguita da un valore appropriato in base al tipo restituito che è stato dichiarato. Ecco un esempio di un metodo della classe `Counter` che restituisce un valore:

```
public int getCount() {
    return count;
}
```

In Java, un metodo può restituire un solo valore. Per restituire più valori, dobbiamo combinarli in un *oggetto composito*, le cui variabili di esemplare contengano tutti i valori che si vogliono restituire, restituendo, in effetti, un riferimento a tale oggetto composito. Oppure, per ottenere in altro modo l'effetto di "restituire" più risultati, possiamo progettare un metodo che modifichi lo stato interno di un oggetto ricevuto come parametro.

Parametri

I parametri di un metodo, separati da virgole, vengono definiti mediante un elenco racchiuso in una coppia di parentesi tonde che segue il nome del metodo. Un parametro è definito da due elementi, il suo tipo e il suo nome. Se un metodo non ha parametri, si usa una coppia di parentesi vuote.

In Java, tutti i parametri vengono passati *per valore*, nel senso che, ogni volta che un parametro viene passato a un metodo, viene fatta una copia del suo valore, che verrà utilizzata all'interno del corpo del metodo. Quindi, se passiamo a un metodo una variabile di tipo `int`, verrà copiato il valore di tale variabile (che è un numero intero). Il metodo può modificare la copia ma non l'originale. Se, invece, passiamo a un metodo come parametro un riferimento a un oggetto, anche tale riferimento viene copiato, e ricordiamo che possiamo avere molte diverse variabili che fanno riferimento al medesimo oggetto. Anche in questo caso, assegnare un diverso valore alla variabile riferimento usata all'interno del metodo non modificherà il riferimento che è stato passato come parametro.

A titolo di esempio, immaginiamo di aggiungere questi due metodi a una classe qualsiasi (come `CounterDemo`).

```
public static void badReset(Counter c) {
    c = new Counter(); // assegna un nuovo contatore
                       // alla variabile locale c
}

public static void goodReset(Counter c) {
    c.reset(); // azzerà il contatore passato dall'invocante
}
```

A questo punto ipotizziamo che la variabile `strikes` faccia riferimento a un esemplare di `Counter` esistente in un determinato contesto e che tale contatore abbia il valore 3.

Se invocassimo `badReset(strikes)`, questo non avrebbe *alcun* effetto sull'esemplare di `Counter` a cui fa riferimento `strikes`. Il corpo del metodo `badReset` assegna alla variabile parametro (`locale`) c un nuovo valore, il riferimento a un esemplare di `Counter` creato all'interno del metodo stesso, ma questa azione non modifica lo stato del contatore preesistente che è stato passato dall'invocante (cioè `strikes`).

Al contrario, se invocassimo `goodReset(strikes)`, questo effettivamente azzererebbe il contatore passato dall'invocante. Questo accade perché entrambe le variabili, c e `strikes`, fanno riferimento al medesimo esemplare di `Counter`, per cui, quando viene invocato il metodo `c.reset()`, l'azione che si verifica è identica a `strikes.reset()`.

Definire costruttori

Un *costruttore* è uno speciale tipo di metodo che viene usato per inizializzare un esemplare di una classe che è appena stata creata, in modo che si porti in uno stato iniziale stabile e coerente. Questo risultato si ottiene solitamente inizializzando ciascuna variabile di esemplare dell'oggetto (a meno che il rispettivo valore predefinito non sia già quello desiderato), anche se, in realtà, un costruttore può anche eseguire calcoli più complessi. Ecco la sintassi da seguire per dichiarare un costruttore in Java:

```
[modificatori] nome(tipo1, param1, ..., tipon, paramn) {
    // corpo del costruttore...
}
```

I costruttori sono, quindi, definiti in modo molto simile agli altri metodi della classe, con alcune differenze che è bene mettere in evidenza:

1. I costruttori non possono essere `static`, `abstract` o `final`, per cui gli unici modificatori che sono consentiti sono quelli che riguardano la loro visibilità (cioè `public`, `protected`, `private` o la visibilità predefinita, a livello di pacchetto).
2. Il nome di un costruttore deve essere uguale al nome della classe in cui è definito. Ad esempio, nella definizione della classe `Counter`, i costruttori devono ugualmente chiamarsi `Counter`.
3. In un costruttore non si specifica il valore restituito (e non si scrive neanche `void`), né, in effetti, il suo corpo restituisce esplicitamente alcunché. Quando l'utilizzatore di una classe ne crea un esemplare, usando una sintassi come questa:

```
Counter d = new Counter(5);
```

l'operatore `new` ha il compito di restituire all'invocante un riferimento al nuovo esemplare appena creato, mentre il metodo costruttore ha solamente la responsabilità di inizializzare lo stato di tale nuovo esemplare.

Una classe può avere molti costruttori, ciascuno dei quali deve avere una diversa *firma*, cioè deve essere distinguibile dagli altri per il tipo e il numero dei parametri che riceve. Se non viene definito esplicitamente alcun costruttore, Java dota la classe di un *costruttore predefinito* implicito, che non richiede argomenti e lascia tutte le variabili di esemplare inizializzate

ai rispettivi valori predefiniti. Se, però, una classe definisce almeno un costruttore, tale costruttore predefinito e implicito non esiste.

Ad esempio, la nostra classe Counter definisce i due seguenti costruttori:

```
public Counter() { }
public Counter(int initial) { count = initial; }
```

Il primo di questi ha un corpo vuoto, {}, per cui il suo obiettivo consiste nella creazione di un contatore con valore iniziale uguale a zero, che è il valore predefinito della variabile di esemplare count. Ciò nonostante, è importante che questo costruttore, seppur banale, sia stato definito in modo esplicito, perché altrimenti la classe non avrebbe avuto un costruttore privo di parametri, dal momento che possiede un costruttore esplicito che impedisce la presenza del costruttore predefinito implicito. In tal caso, l'utilizzatore della classe non avrebbe potuto usare la sintassi `new Counter()`.

La parola chiave this

All'interno del corpo di un metodo non statico, in Java viene automaticamente definita la parola chiave `this`, come riferimento all'esemplare con il quale il metodo è stato invocato: se l'invocante usa la sintassi `thing.foo(a, b, c)`, allora all'interno del metodo `foo`, durante quell'invocazione, la parola chiave `this` fa riferimento all'oggetto che, nel contesto dell'invocante, si chiama `thing`. Sono tre i casi più frequenti che rendono utile la presenza di questo riferimento all'interno del corpo di un metodo:

1. Per memorizzare il riferimento in una variabile o per passarlo come parametro a un altro metodo che si aspetta di ricevere come argomento un esemplare di quel tipo.
2. Per distinguere tra una variabile locale e una variabile di esemplare che hanno lo stesso nome. Se all'interno di un metodo viene dichiarata una variabile locale che ha lo stesso nome di una variabile di esemplare di quella classe, nel codice del metodo tale nome farà riferimento alla variabile locale (infatti diciamo che la variabile locale *maschera o mette in ombra* la variabile di esemplare). In tal caso, si può comunque accedere alla variabile di esemplare usando esplicitamente la notazione “punto”, con il qualificatore `this`. Ad esempio, alcuni programmati sono soliti usare, per i costruttori, questo stile, con i parametri che hanno lo stesso nome della corrispondente variabile di esemplare che vanno a inizializzare:

```
public Counter(int count) {
    this.count = count; // assegna alla variabile di esemplare
                       // il valore del parametro
}
```

3. Per consentire al corpo di un costruttore di invocare il corpo di un altro costruttore. Quando un metodo di una classe invoca un altro metodo di quella stessa classe operando sul medesimo esemplare, viene solitamente utilizzato il nome dell'altro metodo, senza aggiungere alcuna ulteriore specifica (si usa, cioè, il nome del metodo “non qualificato” in altro modo). La sintassi per l'invocazione di un costruttore è, però, particolare: Java consente, all'interno del corpo di un costruttore, l'uso della parola chiave `this` come se fosse un metodo, in modo da poter invocare un altro costruttore, che abbia una diversa firma.

Spesso questo si rivela utile perché tutte le fasi di inizializzazione presenti all'interno di un costruttore possono così essere riutilizzate con valori opportuni dei parametri. Per dare una semplice dimostrazione della sintassi che va utilizzata, possiamo realizzare in modo alternativo la versione priva di argomenti del costruttore di Counter, in modo che invochi la versione che riceve un argomento, passando il valore 0 come parametro esplicito. Scriviamo così:

```
public Counter() {
    this(0); // invoca il costruttore con un parametro passando 0
}
```

Nel Paragrafo 1.7 illustreremo in modo più significativo questa tecnica, applicata a un esempio relativo alla classe CreditCard.

Il metodo main

Alcune classi Java, come la classe Counter, sono pensate per essere utilizzate da altre classi, non per costituire un programma a sé stante. In Java, il flusso di controllo principale di un'applicazione deve partire da una qualche classe, con l'esecuzione di un metodo speciale, che si chiama `main` e deve essere dichiarato in questo modo:

```
public static void main(String[] args) {
    // corpo del metodo main
}
```

Il parametro `args` è un array di oggetti di tipo `String`, cioè è una raccolta indicizzata di stringhe (che sono sequenze di caratteri), la prima delle quali è `args[0]`, la seconda `args[1]` e così via (nel Paragrafo 1.3 parleremo di stringhe e array più approfonditamente). Queste informazioni costituiscono quelli che vengono solitamente detti *argomenti sulla riga dei comandi*, forniti dall'utente del programma quando questo viene messo in esecuzione.

I programmi Java possono essere invocati sulla riga dei comandi usando, appunto, il comando `java` (all'interno di una finestra di comando, o *shell*, di Windows, Linux o Unix), seguito dal nome della classe Java di cui vogliamo eseguire il metodo `main` e da altri argomenti opzionali. Ad esempio, per eseguire il metodo `main` di una classe che si chiama `Aquarium`, dovremo inviare al sistema operativo il comando seguente:

```
java Aquarium
```

In questo caso, l'ambiente di esecuzione di Java (*Java runtime system*) cercherà una versione compilata della classe `Aquarium`, per poi invocarne lo speciale metodo `main`.

Se avessimo definito il programma `Aquarium` in modo che riceva un argomento opzionale che indica il numero di pesci presenti nell'acquario, allora potremmo invocarlo scrivendo nella *shell* questo comando:

```
java Aquarium 45
```

per specificare che vogliamo un acquario contenente 45 pesci. In questo caso, `args[0]` farebbe riferimento alla stringa "45" e sarebbe compito del corpo del metodo `main` interpretare tale stringa come numero desiderato di pesci.

I programmatori che usano un ambiente di sviluppo integrato (**IDE, integrated development environment**), come Eclipse, possono specificare proprio tramite tale ambiente anche gli argomenti opzionali da fornire sulla riga dei comandi nel momento in cui viene eseguito un programma.

Collaudo di unità

Quando si definisce una classe, come Counter, che è destinata a essere utilizzata da altre classi piuttosto che come programma a sé stante, non c'è bisogno di definirvi un metodo `main`. Ciò nonostante, è comodo e utile, in Java, definire comunque tale metodo, per consentire il collaudo della classe isolata da altre, ben sapendo che il metodo non verrà eseguito se non invocando specificatamente il comando `java` su quella classe a sé stante. Naturalmente, per un collaudo più organico, è sempre preferibile usare infrastrutture specifiche, come JUnit.

1.3 Stringhe, involucri, array ed enumerazioni

La classe String

In Java, il tipo di dato fondamentale `char` memorizza un valore che rappresenta un singolo **carattere** e l'insieme dei possibili caratteri, detto **alfabeto**, è l'insieme di caratteri internazionali Unicode, una codifica a 16 bit che copre i linguaggi scritti più utilizzati (alcuni linguaggi di programmazione usano l'insieme di caratteri ASCII, più piccolo, che è un sottoinsieme proprio dell'alfabeto Unicode ed è basato su una codifica a 7 bit). Per descrivere un carattere letterale, in Java si usa una coppia di singoli apici, come '`G`'.

Dato che nei programmi si usano frequentemente sequenze di caratteri (ad esempio, per interagire con gli utenti o per elaborare dati), Java aiuta i programmatori mettendo a disposizione la **classe String**. Un esemplare di stringa (o, più semplicemente, "una stringa") rappresenta una sequenza di zero o più caratteri e la classe fornisce un esteso supporto a molti problemi di elaborazione di testi: nel Capitolo 13 prenderemo in esame parecchi degli algoritmi utilizzati. Per ora, metteremo in evidenza soltanto le caratteristiche principali della classe `String`. Nel linguaggio Java, le stringhe letterali vengono racchiuse tra virgolette, quindi potremmo dichiarare e inizializzare un esemplare di `String` in questo modo:

```
String title = "Data Structures & Algorithms in Java";
```

Individuazione di caratteri

Ogni carattere c all'interno di una stringa s può essere individuato mediante un **indice**, che è uguale al numero di caratteri che precedono c in s . Usando questa convenzione, il primo carattere è associato all'indice 0, mentre l'ultimo si trova in corrispondenza dell'indice $n - 1$, dove n è la lunghezza della stringa. Ad esempio, la stringa `title`, appena definita, ha lunghezza 36. Il carattere che si trova in corrispondenza dell'indice 2 è 't' (il terzo carattere), mentre il carattere all'indice 4 è ' ' (il carattere di spaziatura). In Java, la classe `String` fornisce il metodo `length()`, che restituisce la lunghezza di un esemplare di stringa, e il metodo `charAt(k)`, che restituisce il carattere che si trova in corrispondenza dell'indice k .

Concatenazione

L'operazione principale per combinare stringhe tra loro è la *concatenazione*, che prende una stringa *P* e una stringa *Q* e le combina per generare una nuova stringa, *P + Q*, costituita da tutti i caratteri di *P* seguiti da tutti i caratteri di *Q*. In Java, l'operazione “+” esegue, in effetti, la concatenazione ogni volta che agisce su due stringhe anziché su numeri, in questo modo:

```
String term = "over" + "load";
```

Questo enunciato definisce una variabile di nome *term* che fa riferimento a una stringa il cui valore è “overload” (più avanti nel capitolo discuteremo con maggiore dettaglio degli enunciati di assegnazione e delle espressioni come queste).

La classe *StringBuilder*

Una caratteristica importante della classe *String* è l'*immutabilità* dei suoi esemplari: una volta che un esemplare di stringa è stato creato e inizializzato, il suo valore non può essere modificato. Si tratta di un comportamento assolutamente intenzionale, conseguente al progetto della classe, che consente una grande efficienza e ottimizzazione all'interno della Java Virtual Machine (la macchina virtuale Java).

Siccome, però, in Java *String* è una classe, si tratta di un tipo riferimento e non di un tipo fondamentale, quindi a una variabile di tipo *String* può essere assegnato un diverso esemplare di stringa rispetto a quello che già contiene (anche se il contenuto di un esemplare di stringa non può essere modificato), come in questo esempio:

```
String greeting = "Hello";
greeting = "Ciao";           // abbiamo cambiato idea
```

In Java è anche piuttosto frequente usare la concatenazione tra stringhe per costruire una nuova stringa, per poi usarla per sostituire uno degli operandi della concatenazione:

```
greeting = greeting + '!'; // ora è "Ciao!"
```

È però importante ricordare che questa operazione crea un nuovo esemplare di stringa, ri-copiando, durante tale processo, tutti i caratteri della stringa preesistente. Nel caso di stringhe molto lunghe (come, ad esempio, le sequenze di DNA), questo può richiedere un tempo molto lungo (e, infatti, all'inizio del Capitolo 4 faremo esperimenti relativi all'efficienza della concatenazione tra stringhe).

Allo scopo di consentire modifiche più efficienti alle stringhe di caratteri, Java mette a disposizione anche la classe *StringBuilder*, i cui esemplari, in pratica, sono una versione *modificabile* di una stringa. Questa classe contiene alcuni dei metodi di accesso della classe *String*, a cui aggiunge ulteriori metodi tra i quali citiamo i seguenti:

setCharAt(*k*, *c*): Trasforma nel carattere *c* il carattere che si trova all'indice *k*.

insert(*k*, *s*): Inserisce una copia della stringa *s* a partire dall'indice *k* della sequenza, traslando rigidamente verso indici maggiori i caratteri esistenti, in modo da creare lo spazio necessario.

append(*s*): Aggiunge la stringa *s* al termine della sequenza.

reverse(): Inverte la sequenza.

Per entrambe le classi, `String` e `StringBuilder`, se un indice `k` non appartiene all'insieme degli indici validi per la sequenza di caratteri si verifica una condizione di errore.

La classe `StringBuilder` può risultare molto utile e costituisce anche un interessante caso di studio per un corso di algoritmi e strutture dati. Nel Paragrafo 4.1, infatti, studieremo in modo empirico l'efficienza della classe `StringBuilder`, mentre il Paragrafo 7.2.4 si occuperà della teoria sottostante alla sua implementazione.

Classi involucro

Nella libreria di Java, molti algoritmi e strutture dati sono stati progettati specificatamente per funzionare soltanto con oggetti e non con valori di un tipo fondamentale. Per aggirare questo ostacolo, Java definisce una *classe involucro* (*wrapper class*) per ciascun tipo fondamentale: un esemplare di una delle classi involucro è in grado di memorizzare un singolo valore del tipo fondamentale corrispondente. Nella Tabella 1.2 sono elencati i tipi e le corrispondenti classi involucro, con esempi di creazione e utilizzo di oggetti.

Tabella 1.2: Classi involucro in Java. Ciascuna classe è elencata a fianco del tipo fondamentale corrispondente, con esempi di creazione e utilizzo di oggetti. In ogni riga ipotizziamo che la variabile `obj` sia stata dichiarata con il nome della classe corrispondente.

Tipo	Classe	Esempio di creazione	Esempio di accesso
<code>boolean</code>	<code>Boolean</code>	<code>obj = new Boolean(true);</code>	<code>obj.booleanValue()</code>
<code>char</code>	<code>Character</code>	<code>obj = new Character('Z');</code>	<code>obj.charValue()</code>
<code>byte</code>	<code>Byte</code>	<code>obj = new Byte((byte) 34);</code>	<code>obj.byteValue()</code>
<code>short</code>	<code>Short</code>	<code>obj = new Short((short) 100);</code>	<code>obj.shortValue()</code>
<code>int</code>	<code>Integer</code>	<code>obj = new Integer(1045);</code>	<code>obj.intValue()</code>
<code>long</code>	<code>Long</code>	<code>obj = new Long(10849L);</code>	<code>obj.longValue()</code>
<code>float</code>	<code>Float</code>	<code>obj = new Float(3.934F);</code>	<code>obj.floatValue()</code>
<code>double</code>	<code>Double</code>	<code>obj = new Double(3.934);</code>	<code>obj.doubleValue()</code>

“Auto-boxing” e “auto-unboxing”

Java consente anche di effettuare in modo implicito conversioni tra valori di un tipo fondamentale ed esemplari della classe involucro corrispondente, procedure che prendono il nome di *auto-boxing* e *auto-unboxing*.

In qualsiasi contesto in cui sia prevista la presenza di un esemplare di `Integer` (ad esempio, come parametro) è possibile, in alternativa, fornire un valore `k` di tipo `int`, nel qual caso Java “inscatola” automaticamente il valore con un oggetto di tipo `Integer`, invocando implicitamente `new Integer(k)`: si tratta della procedura di *auto-boxing*. Quando, viceversa, è richiesta la presenza di un valore di tipo `int`, si può usare una variabile `v` di tipo `Integer`, nel qual caso Java “estrae” automaticamente il valore invocando implicitamente `v.intValue()`: si tratta della procedura di *auto-unboxing*. Conversioni analoghe sono ovviamente previste per gli altri tipi fondamentali e i rispettivi involucri. Infine, tutte le classi involucro mettono a disposizione metodi per convertire stringhe letterali in propri esemplari, e viceversa. Il Codice 1.4 illustra molte di tali funzionalità.

Codice 1.4: Esempi di utilizzo della classe `Integer`.

```

1 int j = 8;
2 Integer a = new Integer(12);
3 int k = a;           // invocazione implicita di a.intValue()

```

```

4 int m = j + a;           // il valore di a viene estratto prima dell'addizione
5 a = 3 * m;               // il risultato viene incapsulato prima dell'assegnazione
6 Integer b = new Integer("-135"); // il costruttore accetta una stringa
7 int n = Integer.parseInt("2013"); // uso di un metodo statico della classe Integer

```

Array

Nella programmazione, capita frequentemente di dover tenere traccia di una sequenza posizionale di valori o oggetti tra loro correlati. Ad esempio, potremmo voler fare in modo che un videogioco memorizzi i dieci punteggi migliori: invece di usare dieci diverse variabili, sarebbe preferibile usare un unico nome per il gruppo di valori, con indici numerici che facciano riferimento ai singoli punteggi. Analogamente, potremmo dover progettare un sistema informativo sanitario che memorizzi i dati relativi ai pazienti assegnati ai letti (numerati) di un determinato ospedale: anche in questo caso sarebbe preferibile evitare di definire nel programma 200 variabili soltanto perché l'ospedale ha 200 letti.

In casi come questi possiamo programmare in modo più efficiente usando un *array* ("schiera"), che è una raccolta di variabili omogenee (cioè tutte dello stesso tipo) e disposte in una sequenza posizionale ben definita. Ogni variabile o *cella* di un array ha un *indice*, che si riferisce in modo univoco al valore memorizzato in essa: le celle di un array sono numerate con gli indici 0, 1, 2, e così via. Nella Figura 1.3 abbiamo rappresentato un array contenente i dieci punteggi migliori ottenuti in un videogioco.

punteggi	940	880	830	790	750	660	650	590	510	440
	0	1	2	3	4	5	6	7	8	9
	indici									

Figura 1.3: Rappresentazione grafica di un array di dieci punteggi (valori di tipo int) di un videogioco.

Elementi e capacità di un array

Ogni valore memorizzato all'interno di un array è un suo *elemento*. Dato che la lunghezza di un array determina il numero massimo di informazioni che vi possono essere memorizzate, chiameremo *capacità* tale lunghezza. In Java, la lunghezza di un array di nome *a* può essere ispezionata usando la sintassi *a.length*, per cui le celle dell'array *a* sono associate agli indici 0, 1, 2, e così via fino a *a.length-1*. Infine, si può accedere alla cella di indice *k* con la sintassi *a[k]*.

Errori di limiti

Tentare di usare, come indice nell'array *a*, un numero esterno all'intervallo che va da 0 a *a.length-1* è un errore pericoloso: si parla di "riferimento fuori dai limiti". I riferimenti fuori dai limiti sono stati ripetutamente sfruttati dai pirati informatici (*hacker*), usando uno stratagemma che prende il nome di *buffer overflow attack* (attacco mediante trabocco di una zona di memorizzazione), per mettere a repentaglio la sicurezza di sistemi di calcolo che usano programmi scritti in linguaggi diversi da Java, perché in Java, per migliorare la sicurezza, tutti gli indici usati in un array vengono sempre verificati per controllare che non siano fuori dai limiti dell'array stesso. Se in un array viene usato un indice fuori dai limiti,

l'ambiente di esecuzione di Java segnala una condizione d'errore, che prende il nome di `ArrayIndexOutOfBoundsException`. Questa verifica aiuta Java a far sì che un certo numero di problemi di sicurezza, tra i quali gli attacchi appena citati, vengano evitati.

Dichiarare e costruire array

Gli array, in Java, sono oggetti un po' strani, in quanto tecnicamente non appartengono a un tipo fondamentale, ma non sono nemmeno esemplari di una classe. Detto ciò, Java manipola un esemplare di array come qualunque altro oggetto, e le variabili "di tipo array" sono *variabili riferimento*.

Per dichiarare che una variabile (o un parametro) è un array, usiamo una coppia di parentesi quadre vuote, posta subito dopo il tipo degli elementi che verranno memorizzati nell'array, ad esempio in questo modo:

```
int[] primes;
```

Dato che gli array sono di tipo riferimento, questo enunciato dichiara che la variabile `primes` è un riferimento a un array di numeri interi, senza però costruire immediatamente nessun array. Ci sono, infatti, due modi per creare un array.

La prima modalità di creazione di un array prevede di usare un'assegnazione nel momento in cui si dichiara l'array, fornendo a destra dell'uguale un array in formato letterale, con una sintassi di questo tipo:

```
tipoDiElemento[ ] nomeArray = { valoreIniziale0, valoreIniziale1, ..., valoreInizialeN-1 };
```

Il `tipoDiElemento` può essere qualsiasi tipo fondamentale di Java oppure il nome di qualunque classe, mentre `nomeArray` può essere qualunque identificatore valido in Java. I valori iniziali devono essere del tipo previsto per l'array. Ad esempio, in questo modo potremmo inizializzare l'array `primes`, così che contenga i primi dieci numeri primi:

```
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

Quando si usa, come in questo caso, un array in forma letterale, l'array che viene creato ha esattamente la capacità necessaria per memorizzare i valori forniti.

La seconda modalità di creazione di array prevede l'utilizzo dell'operatore `new`. Dato che, però, un array non è un esemplare di una classe, non usiamo la consueta sintassi che invoca il costruttore, bensì questa:

```
new tipoDiElemento[lunghezza]
```

dove `lunghezza` è un numero intero non negativo che specifica la lunghezza dell'array che si crea. L'operatore `new` restituisce un riferimento al nuovo array e tale riferimento verrà solitamente assegnato a una variabile di tipo array. Ad esempio, l'enunciato seguente dichiara una variabile di tipo array di nome `measurements` e le assegna contestualmente un riferimento a un nuovo array di 1000 celle.

```
double[] measurements = new double[1000];
```

Quando un array viene creato usando l'operatore `new`, a tutti i suoi elementi viene automaticamente assegnato il valore predefinito per quel tipo di dato, per cui, se l'elemento è, ad esempio, di tipo numerico, tutte le celle dell'array verranno inizializzate a zero; se, invece, si tratta di un array di tipo `boolean`, tutte le celle conterranno il valore `false` e se gli elementi sono dei riferimenti (come avviene, ad esempio, nel caso di un array di esemplari di `String`), allora tutte le celle sono inizializzate a `null`.

Tipi enumerativi

Fino a qualche anno fa, spesso i programmatore si trovavano a dover definire una serie di valori interi costanti per rappresentare un insieme finito di possibili scelte. Ad esempio, per rappresentare un giorno della settimana, veniva tipicamente dichiarata una variabile `today` di tipo `int`, a cui assegnare il valore 0 per lunedì (`monday`), 1 per martedì (`tuesday`), e così via.

Oggi, uno stile di programmazione un po' migliore suggerisce di definire variabili statiche costanti (cioè con l'attributo `final`) che rappresentino tale associazione di contenuti:

```
static final int MON = 0;
static final int TUE = 1;
static final int WED = 2;
...
```

perché in questo modo diventa possibile scrivere enunciati di assegnazione come `today = TUE`, decisamente migliori di un criptico `today = 1`. Sfortunatamente la variabile `today`, anche con questo secondo stile di programmazione, è ancora dichiarata di tipo `int`, e, nel momento in cui la si memorizza in una variabile di esemplare o la si passa come parametro, può non essere per nulla evidente il fatto che la si voglia usare per rappresentare un giorno della settimana.

Java consente di seguire un approccio decisamente più elegante per rappresentare scelte in un insieme finito, definendo quello che si chiama "tipo enumerativo" o, in breve, `enum`. Si tratta di tipi a cui si può assegnare soltanto un valore che appartenga a un insieme di nomi ben specificato e si dichiarano in questo modo:

```
modificatore enum nome { nomeValore0, nomeValore1, ..., nomeValoreN-1 };
```

dove il `modificatore` può essere `public`, `protected`, `private` o essere assente. Il `nome` di una enumerazione può essere qualunque identificatore valido in Java, e ciascuno degli identificatori di valore, `nomeValorei`, è il nome di uno dei possibili valori che le variabili di questo tipo enumerativo possono assumere. Ognuno di questi nomi di valori può essere un qualunque identificatore valido in Java, ma la convenzione stilistica di Java prevede che si debba trattare di parole scritte con caratteri maiuscoli. Ad esempio, questa potrebbe essere la definizione di un tipo enumerativo per rappresentare i giorni della settimana:

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
```

Dopo essere stato così definito, `Day` diventa un tipo effettivo e all'interno del programma è possibile dichiarare variabili o parametri di tipo `Day`. Una variabile di tale tipo può essere dichiarata in questo modo:

```
Day today;
```

e questo può essere un enunciato che assegna un valore a tale variabile:

```
today = Day.TUE;
```

1.4 Espressioni

Le variabili e le costanti vengono utilizzate all'interno di *espressioni* per definire nuovi valori e per modificare altre variabili. In questo paragrafo analizzeremo in maggiore dettaglio il funzionamento delle espressioni in Java. Le espressioni utilizzano, al proprio interno, *letterali*, *variabili* e *operatori*. Avendo già parlato di variabili, approfondiamo brevemente il ruolo dei letterali, per poi discutere più dettagliatamente degli operatori.

1.4.1 Letterali

Un *letterale* (*literal*) è un valore “costante” che può essere usato all'interno di un'assegnazione o di un'espressione, e Java prevede i seguenti tipi di letterali:

- Il riferimento `null`, che è l'unico letterale di tipo riferimento a oggetto e il suo utilizzo è consentito come riferimento di qualunque tipo.
- Valori booleani: `true` e `false`.
- Numeri interi: i valori numerici scritti nel codice, come `176` o `-52`, sono considerati di tipo `int`, cioè numeri interi rappresentati con 32 bit. Un letterale che debba essere interpretato come valore numerico intero “lungo”, cioè a 64 bit, deve terminare con un carattere “`l`” o “`1`”, come `176L` o `-52L`.
- Numeri in virgola mobile: i valori numerici scritti nel codice con il punto separatore decimale, come `3.1415` e `135.23`, sono considerati di tipo `double`; perché un letterale numerico sia considerato di tipo `float` bisogna terminarlo con “`f`” o “`F`”. I letterali numerici in virgola mobile possono anche essere espressi con la notazione esponenziale, come `3.14E2` o `.19e10`; viene utilizzata la base 10.
- Caratteri: in Java, le costanti di tipo carattere appartengono all'alfabeto Unicode. Solitamente un carattere letterale viene definito come un unico simbolo racchiuso tra singoli apici: ad esempio, ‘`a`’ e ‘`?'` sono letterali di tipo carattere. Oltre a ciò, Java definisce le seguenti costanti speciali di tipo carattere:

<code>'\n'</code>	(newline)	<code>'\t'</code>	(tab)
<code>'\b'</code>	(backspace)	<code>'\r'</code>	(return)
<code>'\f'</code>	(form feed)	<code>'\\'</code>	(backslash)
<code>'\''</code>	(single quote)	<code>'\"'</code>	(double quote)

- Stringhe letterali: sono sequenze di caratteri racchiuse tra virgolette, come “`dogs cannot climb trees`”.

1.4.2 Operatori

In Java, le espressioni prevedono la composizione di letterali e variabili mediante operatori, dei quali faremo qui una panoramica.

Operatori aritmetici

In Java esistono i seguenti operatori aritmetici binari (cioè con due operandi):

- + addizione
- sottrazione
- * moltiplicazione
- / divisione
- % modulo

L'ultimo operatore, chiamato "modulo", è anche noto come "operatore resto", perché è il resto di una divisione intera. Spesso in matematica si indica l'operatore modulo con "mod", definendolo formalmente in questo modo:

$$n \bmod m = r$$

in modo che esista un numero intero q tale che

$$n = mq + r$$

con $0 \leq r < m$.

Java consente anche l'uso del "meno unario" (cioè l'operatore di sottrazione usato con un solo operando), da scrivere prima di un'espressione aritmetica per cambiare il segno. Inoltre, in un'espressione è anche possibile introdurre parentesi (tonde) per alterare l'ordine di valutazione degli operatori. Quando non vengono usate parentesi, Java determina l'ordine di valutazione usando regole di precedenza abbastanza intuitive. Diversamente dal linguaggio C++, Java non consente il sovraccarico degli operatori per le classi.

Concatenazione di stringhe

Se applicato a stringhe, l'operatore + esegue una *concatenazione*, per cui l'esecuzione di questo frammento di codice:

```
String rug = "carpet";
String dog = "spot";
String mess = rug + dog;
String answer = mess + " will cost me " + 5 + " hours!";
```

ha come effetto l'assegnazione alla variabile answer di un riferimento a questa stringa:

"carpetspot will cost me 5 hours!"

Questo esempio mostra anche come Java converta in stringhe valori che non lo siano (come 5), nel momento in cui siano coinvolti in un'operazione di concatenazione.

Operatori di incremento e decremento

Come i linguaggi C e C++, Java fornisce ai programmatore gli operatori di incremento e decremento: nello specifico, esistono l'operatore di incremento unitario (`++`) e di decremento unitario (`--`). Se uno di questi operatori viene prefisso a una variabile, allora a tale variabile viene aggiunta (o sottratta) un'unità e, poi, questo nuovo valore viene utilizzato all'interno dell'espressione. Se, invece, l'operatore viene scritto dopo la variabile, prima ne viene letto e utilizzato il valore nell'espressione, poi tale valore viene modificato. Quindi, per esempio, il seguente frammento di codice:

```
int i = 8;
int j = i++; // j diventa uguale a 8, poi i diventa uguale a 9
int k = ++i; // i diventa uguale a 10, poi k diventa uguale a 10
int m = i--;
int n = 9 + --i; // i diventa uguale a 8, poi n diventa uguale a 17
```

assegna 8 a `j`, 10 a `k`, 10 a `m`, 17 a `n` e, al termine, 8 a `i`, come evidenziato nei commenti.

Operatori logici

Java consente l'utilizzo dei normali operatori di confronto tra numeri:

- < minore di
- <= minore di o uguale a
- == uguale a
- != diverso da
- >= maggiore di o uguale a
- > maggiore di

Il risultato di qualunque di questi confronti è di tipo `boolean`. Gli stessi confronti operano anche tra valori di tipo `char`, nel qual caso le disuguaglianze sono regolate dai sottostanti codici corrispondenti ai caratteri nello standard Unicode.

Per quanto riguarda i riferimenti, è importante sapere che sono definiti solamente gli operatori `==` e `!=`, in modo che l'espressione `a == b` sia vera se e solo se `a` e `b` fanno entrambi riferimento al medesimo oggetto (oppure hanno entrambi il valore `null`). La maggior parte dei tipi di oggetti definisce il metodo `equals`, in modo che `a.equals(b)` restituisca `true` se e solo se `a` e `b` fanno riferimento a esemplari di quella classe che siano considerati "equivalenti" (pur non essendo il medesimo oggetto): ne ripareremo nel Paragrafo 3.5.

Per i valori di tipo `boolean` sono definiti anche i seguenti operatori:

- ! not (prefisso)
- && and condizionale
- || or condizionale

Gli operatori booleani `&&` e `||` non valutano il secondo operando (quello di destra) nel caso in cui il suo valore non sia necessario per determinare il valore dell'operazione. Questa caratteristica di "valutazione in cortocircuito" è molto utile per costruire espressioni booleane nelle quali si inizia verificando se una determinata condizione è verificata (come,

ad esempio, il fatto che un indice in un array sia valido), per poi proseguire con la verifica di una condizione che, se la prima fosse fallita, darebbe luogo a una situazione di errore.

Operatori bit per bit

In Java sono presenti anche i seguenti operatori che agiscono bit per bit su valori booleani o numerici interi:

- complemento bit per bit (operatore unario prefisso)
- & and bit per bit
- | or bit per bit
- ^ or esclusivo bit per bit
- << scorrimento dei bit verso sinistra, inserendo degli zeri
- >> scorrimento dei bit verso destra, inserendo duplicati del bit di segno
- >>> scorrimento dei bit verso destra, inserendo degli zeri

L'operatore di assegnazione

L'operatore di assegnazione (o assegnamento) in Java è “=” e viene usato per assegnare un valore a una variabile, secondo questa sintassi:

variabile = espressione

dove *variabile* è una qualsiasi variabile a cui si possa fare riferimento dall'interno del blocco di enunciati che contiene l'espressione. Il valore assunto da un'operazione di assegnazione è il valore dell'espressione usata, appunto, nell'assegnazione, per cui se j e k sono due variabili di tipo `int`, un enunciato di assegnazione come questo è assolutamente corretto:

```
j = k = 25; // funziona perché gli operatori '=' sono valutati  
// da destra a sinistra
```

Operatori di assegnazione composti

Oltre al normale operatore di assegnazione (-), Java consente anche l'utilizzo di un certo numero di altri operatori di assegnazione che combinano un'operazione binaria e un'assegnazione, in questa forma:

variabile op = espressione

dove *op* è un operatore binario qualsiasi. L'espressione appena riportata è, in generale, equivalente a:

variabile = variabile op espressione

per cui $x *= 2$ è equivalente a $x = x * 2$. Se, però, la *variabile* contiene a sua volta un'espressione (ad esempio, un indice in un array), l'espressione viene valutata una volta sola, per cui il seguente frammento di codice:

```
a[5] = 10;
j = 5;
a[j++] += 2; // NON equivale a a[j++] = a[j++] + 2
```

assegna 12 alla cella `a[5]` e 6 a `j`.

Precedenza tra gli operatori

Java assegna agli operatori dei livelli di precedenza, che determinano l'ordine in cui vengono eseguite le operazioni quando l'assenza di parentesi genererebbe ambiguità. Ci serve, ad esempio, una regola per decidere in che modo vada valutata l'espressione "5+2*3": il suo valore è 21 o 11? Java dice che è 11. La Tabella 1.3 riporta le precedenze tra gli operatori in Java (incidentalmente, sono le stesse usate dai linguaggi C e C++).

Tabella 1.3: Le regole di precedenza in Java. In Java, gli operatori vengono valutati secondo l'ordine qui indicato, a meno che non vengano usate parentesi (tonde) per alterarlo.

Operatori che figurino sulla stessa riga della tabella sono valutati da sinistra a destra (ad eccezione degli operatori prefissi e di assegnazione, che sono valutati da destra a sinistra), con il vincolo dell'esecuzione condizionale che caratterizza le operazioni booleane `&&` e `||`. Le operazioni sono elencate per precedenza decrescente, con `espr` che indica un'espressione elementare o tra parentesi. In mancanza di parentesi, gli operatori con precedenza più elevata vengono eseguiti prima di quelli con precedenza inferiore.

Precedenza tra gli operatori	
Nome	Simboli
1 indice array	<code>[]</code>
invocazione metodo	<code>()</code>
operatore punto	<code>.</code>
2 operatori postfissi	<code>espr++ espr--</code>
operatori prefissi	<code>++espr --espr +espr -espr ^espr !espr</code>
cast	<code>(tipo) espr</code>
3 molt./div.	<code>* / %</code>
4 add./sottr.	<code>+ -</code>
5 scorrimento	<code><< >> >>></code>
6 confronto	<code>< <= > >= instanceof</code>
7 uguaglianza	<code>== !=</code>
8 and bit per bit	<code>&</code>
9 or escl. bit per bit	<code>^</code>
10 or bit per bit	<code> </code>
11 and	<code>&&</code>
12 or	<code> </code>
13 condizionale	<code>esprBooleana ? valoreSeVera : valoreSeFalsa</code>
14 assegnazione	<code>= += -= *= /= %= <<= >>= &= ^= =</code>

Abbiamo già parlato di quasi tutti gli operatori elencati nella Tabella 1.3, con l'evidente eccezione dell'operatore condizionale che prevede la valutazione di un'espressione booleana, per poi assumere il valore appropriato in relazione al fatto che tale espressione sia risultata vera o falsa (infine, nel prossimo capitolo parleremo dell'operatore `instanceof`).

1.4.3 Conversioni di tipo

Il *cast* (letteralmente, “forzatura”) è un’operazione che consente di cambiare tipo a un valore: in pratica, possiamo prendere un valore di un tipo e *farlo* a diventare un valore equivalente di un altro tipo. In Java esistono due forme di cast: *esplicito* e *implicito*.

Cast esplicito

Java consente al programmatore di effettuare un cast esplicito usando questa sintassi:

(tipo) espressione

dove *tipo* è il tipo che si vuole attribuire alla *espressione*. Questa sintassi può essere utilizzata soltanto per un cast tra due diversi tipi fondamentali, oppure tra un riferimento di un tipo e un riferimento di un altro tipo: qui parleremo del cast tra tipi fondamentali, rimandando al Paragrafo 2.5.1 la conversione esplicita tra riferimenti.

Il cast che consente di trasformare un valore di tipo `int` in un valore di tipo `double` viene chiamato “conversione con ampliamento” (*widening cast*), perché il tipo `double` è più “capiente” del tipo `int` e una tale conversione può, quindi, avvenire senza perdita di contenuto informativo. Al contrario, un cast che trasformi un valore di tipo `double` in un valore di tipo `int` è una “conversione con restrizione” (*narrowing cast*) e può avere come conseguenza una perdita di informazione, dal momento che un’eventuale parte frazionaria del valore convertito sarà troncata. Consideriamo, ad esempio, queste conversioni:

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1;           // i1 assume il valore 3
int i2 = (int) d2;           // i2 assume il valore 3
double d3 = (double) i2;    // d3 assume il valore 3.0
```

Sebbene un cast esplicito non sia in grado di convertire direttamente un valore di un tipo fondamentale in un riferimento, né viceversa, tali conversioni di tipo si possono eseguire in altro modo. Nel Paragrafo 1.3 abbiamo già parlato delle conversioni tra i tipi fondamentali di Java e le corrispondenti classi involucro (ad esempio, tra `int` e `Integer`): per comodità, tali classi involucro mettono a disposizione, tra le altre cose, metodi statici che effettuano una conversione tra un valore del corrispondente tipo fondamentale e un valore di tipo `String`.

Ad esempio, il metodo `Integer.toString` accetta un parametro di tipo `int` e restituisce una rappresentazione di tale valore sotto forma di oggetto di tipo `String`, mentre il metodo `Integer.parseInt` accetta come parametro una stringa e restituisce, come tipo `int`, il valore da essa rappresentato (se la stringa non rappresenta un numero intero, viene lanciata un’eccezione di tipo `NumberFormatException`). Ecco un esempio del loro utilizzo:

```
String s1 = "2014";
int i1 = Integer.parseInt(s1);    // i1 assume il valore 2014
int i2 = -35;
String s2 = Integer.toString(i2); // s2 assume il valore "-35"
```

Le altre classi involucro, come `Double`, dispongono di metodi assolutamente analoghi.

Cast implicito

Ci sono casi in cui Java esegue un *cast implicito*, in base al contesto di un'espressione. Come esempio, una “conversione con ampliamento” (che, quindi, non possa provocare perdita di informazione) può essere effettuata tra due tipi primitivi (ad esempio, per trasformare in `double` un valore di tipo `int`) senza usare in modo esplicito l'operatore di cast. Se, invece, si prova a scrivere codice che richiederebbe una implicita “conversione con restrizione”, si otterrà una segnalazione di errore da parte del compilatore. L'esempio seguente mostra enunciati di assegnazione che contengono un cast implicito lecito e uno non lecito:

```
int i1 = 42;
double d1 = i1; // d1 assume il valore 42.0
i1 = d1;        // errore in compilazione: possibile perdita di precisione
                // segnalata come "possible loss of precision"
```

Il cast implicito si verifica anche quando si eseguono operazioni aritmetiche che coinvolgono tipi di dati numerici diversi, in particolare quando si esegue un'operazione tra un operando di tipo intero e un operando in virgola mobile: in tal caso, prima di eseguire l'operazione il valore intero viene implicitamente convertito nel corrispondente valore in virgola mobile. Ad esempio, l'espressione `3 + 5.7` viene implicitamente convertita in `3.0 + 5.7`, prima di calcolare il valore risultante, `8.7`, di tipo `double`.

Capita piuttosto spesso di combinare un cast esplicito e un cast implicito per eseguire una divisione in virgola mobile tra due operandi di tipo intero. L'espressione `(double) 7 / 4` produce come risultato `1.75`, perché la precedenza tra operatori impone che il cast venga eseguito per primo, come se si fosse scritto `((double) 7) / 4`; poi, `7.0 / 4` diventa implicitamente uguale a `7.0 / 4.0`. Si noti che, invece, l'espressione `(double) (7 / 4)` ha come risultato `1.0`.

Per inciso, esiste una situazione, in Java, in cui il cast è consentito soltanto in modo implicito: la concatenazione tra stringhe. Ogni volta che una stringa viene concatenata con un oggetto di qualsiasi tipo o con un valore di un tipo fondamentale, tale oggetto o valore viene automaticamente convertito in una stringa, per poi effettuare la concatenazione, ma il cast esplicito non è ammesso. Di conseguenza, questi enunciati di assegnazione non sono corretti:

```
String s = 22;                      // sbagliato!
String t = (String) 4.5;            // sbagliato!
String u = "Value = " + (String) 13; // sbagliato!
```

Per effettuare, in tali casi, una conversione che generi una stringa, dobbiamo usare il metodo `toString` adeguato, oppure eseguire il cast implicito mediante l'operazione di concatenazione con una stringa vuota. Gli enunciati seguenti sono, quindi, corretti:

```
String s = Integer.toString(22); // corretto
String t = "" + 4.5;           // corretto, ma uno stile pessimo
String u = "Value = " + 13;    // corretto
```

1.5 Controllo di flusso

Il controllo di flusso in Java è simile a quello utilizzato in altri linguaggi di alto livello. In questo paragrafo vedremo una panoramica delle strutture su cui si basa il controllo di flusso: gli enunciati **if** e **switch**, i cicli, la terminazione dei metodi e le limitate forme di "salti" disponibili (cioè gli enunciati **break** e **continue**).

1.5.1 Gli enunciati if e switch

In Java, gli enunciati che controllano l'esecuzione condizionata funzionano in modo assolutamente analogo a quanto avviene in altri linguaggi e forniscono strumenti che consentono di prendere una decisione, eseguendo poi uno o più diversi blocchi di enunciati sulla base del risultato di tale decisione.

L'enunciato if

La sintassi di un semplice enunciato **if** è la seguente:

```
if (espressioneBooleana)
    corpoSeVera
else
    corpoSeFalsa
```

dove l'*espressioneBooleana* è, appunto, un'espressione booleana, mentre *corpoSeVera* e *corpoSeFalsa* sono, ciascuno, un singolo enunciato oppure un blocco di enunciati racchiusi da una coppia di parentesi graffe. Si noti che, diversamente da quanto accade in altri linguaggi simili, in Java il valore di controllo di un enunciato **if** deve essere un'espressione booleana e, in particolare, non può in alcun modo essere un'espressione il cui valore sia un numero intero. Invece, come in altri linguaggi, in un enunciato **if** la sezione **else** (e il *corpoSeFalsa* ad essa associato) è facoltativa. Si possono anche raggruppare insieme più condizioni booleane, in questo modo:

```
if (primaEspressioneBooleana)
    primoCorpo
else if (secondaEspressioneBooleana)
    secondoCorpo
else if (terzaEspressioneBooleana)
    terzoCorpo
else
    ultimoCorpo
```

Se la *primaEspressioneBooleana* è falsa, verrà verificata la *secondaEspressioneBooleana*, e così via. Un enunciato **if** può avere un numero arbitrario di sezioni **else if**, e si possono usare le parentesi graffe per definire l'estensione di alcuni dei relativi corpi, o anche di tutti.

Come semplice esempio, vediamo il controllore di un robot, che potrebbe essere caratterizzato dalla logica seguente:

```
if (door.isClosed()) // se la porta è chiusa...
    door.open();      // apri la porta
    advance();        // procedi
```

Come si può notare, il comando conclusivo, `advance()`, non fa parte del corpo la cui esecuzione è condizionata: verrà, quindi, eseguito incondizionatamente (dopo aver aperto un'eventuale porta chiusa).

È possibile, poi, “annidare” una struttura di controllo dentro l’altra, usando, se necessario, le parentesi graffe per rendere chiara l’estensione dei vari corpi presenti. Tornando all’esempio del robot, vediamo una logica di controllo un po’ più complicata, che tiene conto della eventuale azione di apertura della serratura di una porta chiusa:

```
if (door.isClosed()) { // se la porta è chiusa...
    if (door.isLocked()) // se la porta è chiusa a chiave...
        door.unlock();   // apri la serratura
        door.open();      // apri la porta
    }
    advance();           // procedi
```

La logica che governa questo esempio può essere schematicamente rappresentata da un cosiddetto *diagramma di flusso* (*flowchart*), che si può analizzare nella Figura 1.4.

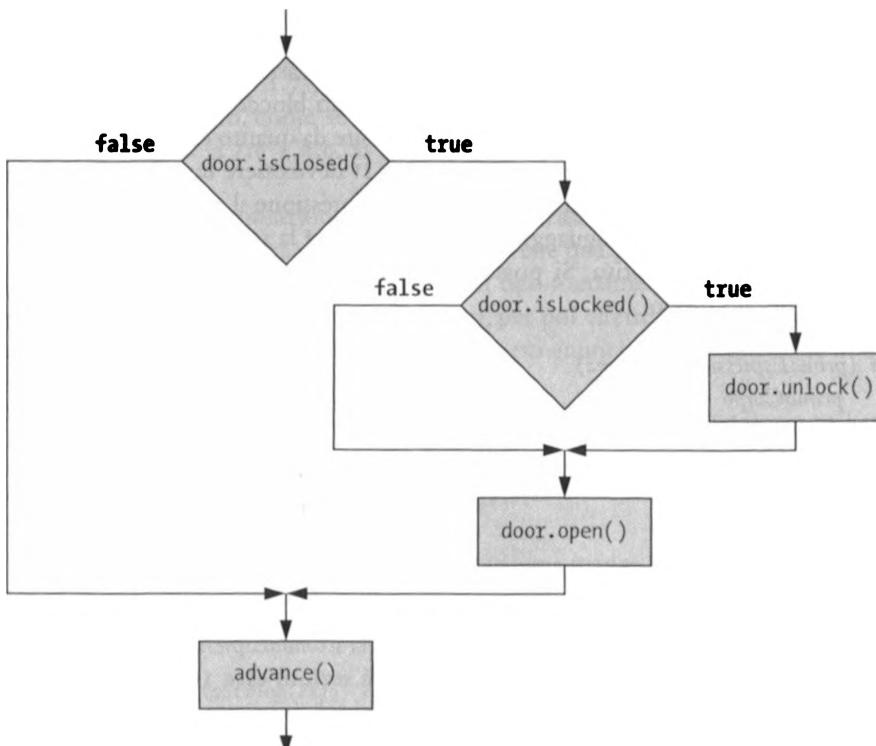


Figura 1.4: Un diagramma di flusso che descrive la logica di un programma mediante enunciati condizionali annidati.

Quello che segue, infine, è un esempio di clausole **if** e **else** annidate:

```
if (snowLevel < 2) { // se è nevicato poco
    goToClass();      // vai a scuola
    comeHome();        // poi, torna a casa
} else if (snowLevel < 5) { // altrimenti, se è nevicato un po' di più
    goSledding();     // vai in giro con la slitta
    haveSnowballFight(); // poi, tira palle di neve
} else                  // altrimenti (cioè, se è nevicato veramente tanto)
    stayAtHome();       // stai a casa
                                // in un corpo con un solo enunciato non servono graffe
```

Enunciati **switch**

Java consente di effettuare un controllo di flusso basato su più valori usando l'enunciato **switch**, che risulta essere particolarmente utile con i tipi enumerativi. Nel seguito proponiamo un esempio piuttosto significativo, basato su una variabile, **d**, di tipo **Day**, il tipo enumerativo definito nel Paragrafo 1.3.

```
switch (d) {
    case MON:
        System.out.println("This is tough.");
        break; // lunedì: è dura...
    case TUE:
        System.out.println("This is getting better.");
        break; // martedì: va un po' meglio...
    case WED:
        System.out.println("Half way there.");
        break; // mercoledì: siamo a metà...
    case THU:
        System.out.println("I can see the light.");
        break; // giovedì: si vede la luce in fondo al tunnel...
    case FRI:
        System.out.println("Now we are talking.");
        break; // venerdì: finalmente se ne può parlare...
    default:
        System.out.println("Day off!"); // è finita!
}
```

L'enunciato **switch** valuta un'espressione che abbia come valore un numero intero, una stringa o una costante enumerativa, facendo poi in modo che il flusso dell'esecuzione salti direttamente alla sezione di codice etichettata con il valore di tale espressione; se non c'è l'etichetta cercata, il controllo passa alla sezione contrassegnata come "**default**". Quello appena descritto è, però, l'unico "salto" esplicito messo in atto dall'enunciato **switch**, per cui il flusso dell'esecuzione "scivola" nella sezione successiva del codice, a meno che la sezione stessa non termini con un enunciato **break** (che costringe il flusso a saltare alla fine dell'enunciato **switch**).

1.5.2 Cicli

Un altro importante meccanismo di controllo del flusso d'esecuzione, in un linguaggio di programmazione, è il ciclo o iterazione. Java mette a disposizione del programmatore tre diversi tipi di ciclo.

Cicli while

Il tipo di ciclo più semplice, in Java, è il ciclo `while`: si tratta di un ciclo che verifica se una determinata condizione è soddisfatta ed esegue il corpo del ciclo ogni volta che tale condizione risulta essere *vera*. La sintassi per questo tipo di reiterata esecuzione condizionata di un corpo è la seguente:

```
while (espressioneBooleana)
    corpoDelCiclo
```

Come nell'enunciato `if`, l'*espressioneBooleana* può essere un'espressione booleana qualsiasi, mentre il *corpoDelCiclo* è un altrettanto arbitrario blocco di codice (che può contenere al proprio interno altre strutture di controllo del flusso, che vengono di nuovo dette "annidate"). L'esecuzione di un ciclo `while` inizia con la valutazione dell'espressione booleana che lo controlla: se il risultato di tale valutazione è `true`, il corpo del ciclo viene eseguito. Al termine di ciascuna esecuzione del corpo, la condizione di controllo del ciclo viene valutata di nuovo e, se il risultato è `true`, viene eseguita un'altra iterazione del corpo. Se, invece, il risultato della valutazione è `false` (ammesso che ciò prima o poi avvenga), il ciclo termina e il flusso dell'esecuzione esce dal ciclo, per proseguire con l'istruzione immediatamente successiva al corpo del ciclo.

Come esempio, vediamo un ciclo che fa avanzare un indice all'interno di un array di nome `data` fino al momento in cui trova un dato il cui valore sia uguale a `target`, oppure ha raggiunto la fine dell'array:

```
int j = 0;
while ((j < data.length) && (data[j] != target))
    j++;
```

Quando il ciclo termina, il valore della variabile `j` sarà uguale all'indice della cella più a sinistra tra quelle che contengono `target`, se questo è presente nell'array, altrimenti `j` sarà uguale alla lunghezza dell'array (un valore che è identificabile come indice non valido e rappresenta il fallimento della ricerca). La correttezza del ciclo si basa sulla valutazione in cortocircuito dell'operatore logico `&&`, come già visto nel Paragrafo 1.4.2. Prima di accedere al valore `data[j]` verifichiamo volutamente che sia `j < data.length`, in modo da garantire la validità dell'indice `j`: se avessimo scritto in ordine inverso tale condizione composta, la valutazione di `data[j]` lancerebbe un'eccezione di tipo `ArrayIndexOutOfBoundsException` ogniqualvolta `target` non fosse presente nell'array (il Paragrafo 2.4 tratterà con maggiore dettaglio le eccezioni).

Osserviamo che un ciclo `while` può non eseguire il proprio corpo, nel caso in cui la sua condizione di controllo sia `false` fin dall'inizio. Nell'esempio precedente, il nostro ciclo non incrementa affatto il valore di `j` in tutti quei casi in cui il valore di `data[0]` è proprio uguale a `target` (oppure l'array ha lunghezza zero).

Cicli do-while

Java prevede un'altra forma di ciclo `while`, che verifica la condizione booleana di controllo al *termine* di ciascuna esecuzione del corpo del ciclo, invece che all'inizio. Questa forma prende il nome di ciclo `do-while` e questa è la sua sintassi:

do

corpoDelCiclo
while (*espressioneBooleana*)

Come prima conseguenza della natura del ciclo **do-while** c'è il fatto che il suo corpo viene sempre eseguito almeno una volta (diversamente dal ciclo **while**, il cui corpo può "essere eseguito zero volte" se la condizione di controllo è inizialmente falsa). Questa struttura iterativa è particolarmente utile in quelle situazioni in cui la condizione di controllo non è ben definita finché il corpo non è stato eseguito almeno una volta. Consideriamo come esempio la richiesta di dati in ingresso forniti dall'utente di un programma, per poi elaborare in qualche modo ciò che viene ricevuto (nel Paragrafo 1.6 vedremo con maggiore dettaglio come, in un programma Java, si possano gestire le informazioni in ingresso, *input*, e in uscita, *output*). In tal caso, una possibile condizione di terminazione del ciclo potrebbe essere l'inserimento di una stringa vuota da parte dell'utente: anche in tal caso, però, vogliamo gestire il dato fornito in ingresso e informare l'utente che ha scelto di uscire dal programma. L'esempio che segue mostra come si possa risolvere il problema:

```
String input;
do {
    input = getInputString(); // chiede all'utente di fornire una stringa
    handleInput(input);      // elabora la stringa
} while (input.length() > 0);
```

Cicli for

Il terzo tipo di ciclo previsto da Java è il ciclo **for**, che esiste in due diverse forme. La prima, che chiameremo "tradizionale", ha una sintassi molto simile a quella dei cicli **for** nei linguaggi C e C++, mentre la seconda, che viene spesso chiamata "ciclo *for-each*" (cioè "per ogni"), è stata inserita nel linguaggio Java nel 2004, con la versione SE 5. Questa seconda forma ha una sintassi più concisa ed è relativa alla scansione iterativa degli elementi di un array o di contenitori di altri tipi, purché siano adeguatamente predisposti.

La sintassi del ciclo **for** tradizionale prevede quattro sezioni (ciascuna delle quali può, però, essere vuota): una inizializzazione, un'espressione booleana che funge da condizione di controllo, un enunciato di incremento (o, meglio, di modifica) e il corpo. Ecco la struttura:

for (*inizializzazione; espressioneBooleana; incremento*)
corpoDelCiclo

Uno degli utilizzi più frequenti del ciclo **for** riguarda l'iterazione basata su un numero intero che agisce da indice, come in questo esempio:

```
for (int j = 0; j < n; j++)
    // fai qualcosa
```

Il comportamento di un ciclo **for** è molto simile a quello del seguente ciclo **while**, equivalente:

```
{
    inizializzazione;
    while (espressioneBooleana) {
        corpoDelCiclo;
```

```
    incremento;
}
```

La sezione di *inizializzazione* verrà eseguita una sola volta, prima che inizi la restante parte del ciclo, e viene solitamente utilizzata per inizializzare variabili preesistenti oppure per dichiarare e contestualmente inizializzare nuove variabili. Va ricordato che qualunque variabile che venga dichiarata nella sezione di inizializzazione sarà visibile solamente per la durata dell'esecuzione del ciclo **for**.

L'*espressione Booleana* verrà valutata immediatamente prima di ciascuna possibile iterazione del ciclo e va progettata in modo analogo a quella di un ciclo **while**, nel senso che se il suo valore è **true** il corpo del ciclo viene eseguito, altrimenti (cioè se il suo valore è **false**) il ciclo termina e il programma prosegue con l'enunciato immediatamente successivo al corpo del ciclo **for**.

La sezione di *incremento* viene eseguita subito dopo ciascuna iterazione del vero e proprio corpo del ciclo e viene tradizionalmente utilizzata per aggiornare il valore della principale variabile di controllo del ciclo. Ciò detto, però, l'enunciato di *incremento* può, in realtà, essere un enunciato valido qualsiasi, consentendo una notevole flessibilità nella scrittura del codice.

Come esempio concreto, vediamo un metodo che calcola la somma dei valori di tipo **double** contenuti in un array usando un ciclo **for**:

```
public static double sum(double[] data) {
    double total = 0;
    for (int j = 0; j < data.length; j++) // notare l'uso di length
        total += data[j];
    return total;
}
```

Come ulteriore esempio, il metodo seguente individua il valore massimo all'interno di un array (non vuoto):

```
public static double max(double[] data) {
    double currentMax = data[0]; // ipotizza che il primo sia il massimo
    for (int j = 1; j < data.length; j++) // analizza tutti gli altri
        if (data[j] > currentMax) // se data[j] è il massimo fin qui...
            currentMax = data[j]; // lo memorizza come attuale massimo
    return currentMax;
}
```

Si noti come un enunciato condizionale (**if**) sia stato annidato all'interno del corpo del ciclo senza che sia stato necessario usare una coppia di parentesi graffe per definire, appunto, il corpo del ciclo, dal momento che l'intero costrutto condizionale è, dal punto di vista sintattico, un singolo enunciato.

Ciclo **for-each**

Dato che la scansione degli elementi di una raccolta di dati, come un array, è una situazione molto frequente, Java prevede una notazione abbreviata per un ciclo di questo tipo, detto *ciclo for-each* ("per ogni"). La sintassi è la seguente:

```
for (tipoDiElemento nome : contenitore)
    corpoDelCiclo
```

dove il *contenitore* è un array di tipo *tipoDiElemento* (oppure una raccolta che implementa l'interfaccia *Iterable*, come vedremo nel Paragrafo 7.4.1).

Rivedendo uno degli esempi precedenti, il ciclo tradizionalmente utilizzato per calcolare la somma degli elementi di un array contenente valori di tipo *double* può essere scritto in questo modo:

```
public static double sum(double[] data) {
    double total = 0;
    for (double val : data) // ciclo Java di tipo for-each
        total += val;
    return total;
}
```

Quando si usa un ciclo for-each non c'è un utilizzo esplicito di un indice all'interno dell'array: è la variabile *nome* a rappresentare, iterazione dopo iterazione, uno specifico elemento dell'array, anche se, all'interno del corpo del ciclo, non c'è nulla che indichi quale elemento si stia manipolando.

È bene mettere in evidenza che assegnare un valore alla variabile *nome* non ha alcun effetto sul contenuto dell'array sottoposto a scansione, per cui il metodo che segue è semplicemente un tentativo fallito di moltiplicare per uno stesso fattore tutti i valori presenti in un array numerico:

```
public static void scaleBad(double[] data, double factor) {
    for (double val : data)
        val *= factor; // modifica soltanto la variabile locale val
}
```

Per poter sovrascrivere i valori contenuti nelle celle di un array è necessario usare un indice. Il problema precedente, quindi, si risolve con un ciclo **for** tradizionale, come il seguente:

```
public static void scaleGood(double[] data, double factor) {
    for (int j = 0; j < data.length; j++)
        data[j] *= factor; // sovrascrive la cella dell'array
}
```

1.5.3 Enunciati per il controllo di flusso esplicito

Java permette anche l'utilizzo di enunciati che provocano cambiamenti espliciti nel flusso d'esecuzione di un programma.

Terminare l'esecuzione di un metodo

Se, in Java, si dichiara che un metodo non restituisce alcun valore (usando la parola chiave **void**), quando il flusso d'esecuzione raggiunge l'ultima riga di codice del metodo il controllo torna al metodo invocante, così come accade nel momento in cui all'interno del metodo viene eseguito un enunciato **return** (privo di argomenti). Se, invece, si dichiara che un metodo restituisce effettivamente un valore di un certo tipo, il metodo deve terminare

restituendo un valore del tipo richiesto, sotto forma di argomento di un enunciato `return`. Ne consegue che l'enunciato `return` deve essere l'ultimo enunciato eseguito all'interno di un metodo, dal momento che la parte di codice che segue non verrà mai raggiunta.

Si noti che è molto diverso affermare che un enunciato sia l'ultima riga di codice che viene *eseguito* in un metodo piuttosto che l'ultima riga fisicamente scritta nel metodo. L'esempio seguente, che è corretto, illustra bene il principio di funzionamento dell'enunciato `return`:

```
public static double abs(double value) {
    if (value < 0) // value è negativo
        return -value; // per cui restituisce il suo opposto
    return value;   // restituisce il valore originario, non negativo
}
```

Nell'esempio precedente, la riga `return -value;` non è, evidentemente, l'ultima riga scritta nel codice del metodo, ma può essere l'ultima riga che viene eseguita, nel caso in cui il valore di `value` sia negativo. Un tale enunciato interrompe esplicitamente il flusso d'esecuzione del metodo, come fanno altri due enunciati di controllo esplicito, che vengono utilizzati all'interno di cicli e di enunciati `switch`.

L'enunciato `break`

Abbiamo visto per la prima volta l'enunciato `break` nel Paragrafo 1.5.1, dove è stato usato per uscire dal corpo di un enunciato `switch`. Più in generale, lo si può usare per uscire dal corpo del più interno enunciato `switch`, `for`, `while` o `do-while`, se ve ne sono diversi annidati. L'esecuzione di un enunciato `break` porta il flusso del programma alla riga di codice che segue il corpo del ciclo o dell'enunciato `switch` che contiene il `break` stesso.

L'enunciato `continue`

L'enunciato `continue` può essere usato soltanto all'interno del corpo di un ciclo e porta il flusso d'esecuzione a ignorare i successivi enunciati previsti *dall'iterazione in corso*, ma, diversamente dall'enunciato `break`, non fa terminare il ciclo e riporta il controllo all'inizio del ciclo stesso, nell'ipotesi che la condizione di controllo sia rimasta `true`.

1.6 Casi semplici di input/output

Java possiede un ricco insieme di classi e metodi che consentono a un programma di trasferire dati mediante attività di input/output (cioè ingresso/uscita o acquisizione/visualizzazione di dati). In Java sono presenti classi che consentono la progettazione di interfacce utente grafiche, complete di finestre *pop-up* e di menu a discesa (*pull-down*), oltre a metodi che visualizzano o acquisiscono informazioni testuali o numeriche. Senza dimenticare che Java mette a disposizione metodi per gestire oggetti grafici, immagini, suoni, pagine web e eventi prodotti dal *mouse* (come selezioni, trascinamenti e sovrapposizioni). Inoltre, molti di questi metodi di gestione delle attività di input e di output possono essere utilizzati sia in programmi a sé stanti sia in *applet* (che sono programmi Java eseguiti all'interno di un *browser web*).

Sfortunatamente l'analisi dettagliata del funzionamento di questi metodi per la costruzione di elaborate interfacce grafiche per l'interazione con l'utente va ben al di là degli obiettivi di questo testo. Ciò nonostante, per completezza, in questo paragrafo descriviamo come si possano compiere, in Java, le più semplici azioni di acquisizione (*input*) e visualizzazione (*output*) di dati.

Tali semplici attività di input/output avvengono, in Java, tramite la finestra di *console* o *shell*. In relazione all'ambiente Java che si sta utilizzando, tale finestra può essere una speciale finestra *pop-up* utilizzata per visualizzare e acquisire testo, oppure una finestra che consente l'invio di comandi al sistema operativo (e, in quest'ultimo caso, si chiama solitamente *shell*, terminale o finestra di comando).

Semplici metodi di output

In Java esiste un oggetto statico predefinito, chiamato `System.out`, che visualizza informazioni sul dispositivo di output standard, che è la finestra di console in cui il programma Java viene eseguito. La maggior parte dei sistemi operativi consente, tramite la finestra di console, di modificare la destinazione del flusso di uscita, cioè di ridefinire il dispositivo di output standard, in modo che sia un file o anche il flusso di ingresso di un altro programma. L'oggetto `System.out` è un esemplare della classe `java.io.PrintStream`, che definisce metodi per un flusso di uscita con *buffer*. Questo significa che i caratteri che devono essere inviati al dispositivo di output vengono prima inseriti in una zona di memoria temporanea, detta appunto *buffer*, che viene poi svuotata nel momento in cui la finestra di console (o, più in generale, il dispositivo di uscita destinatario delle informazioni) è in grado di visualizzare tali caratteri.

Nello specifico, per effettuare semplici azioni di output la classe `java.io.PrintStream` definisce i metodi seguenti (dove usiamo il termine *tipoFondamentale* per fare riferimento a uno qualsiasi dei possibili tipi di dato fondamentale in Java):

```
print(String s): Visualizza la stringa s.  
print(Object o): Visualizza l'oggetto o usando il suo metodo toString.  
print(tipoFondamentale b): Visualizza il valore b di un tipoFondamentale.  
println(String s): Visualizza la stringa s, seguita dal carattere newline (che "va a capo").  
println(Object o): Come print(o), poi "va a capo".  
println(tipoFondamentale b): Come print(b), poi "va a capo".
```

Un esempio di output

Consideriamo, ad esempio, il seguente frammento di codice:

```
System.out.print("Java values: ");  
System.out.print(3.1416);  
System.out.print(',');  
System.out.print(15);  
System.out.println(" (double,char,int).");
```

L'esecuzione di questo frammento visualizza nella finestra di console di Java le seguenti informazioni:

Java values: 3.1416,15 (double,char,int).

Input semplificato usando la classe `java.util.Scanner`

Così come per visualizzare testo nella finestra di console di Java si utilizza un oggetto speciale, ne esiste un altro, `System.in`, che consente di acquisire dati in ingresso attraverso la medesima finestra. I dati in ingresso vengono acquisiti, dal punto di vista tecnico, tramite il “dispositivo di input standard” che, se non diversamente specificato, è la tastiera del calcolatore: i caratteri che vengono digitati, oltre che acquisiti dal programma, risultano anche ricopiatati (mediante “eco”) nella finestra di console. L’oggetto `System.in` è, appunto, associato a tale dispositivo di input standard. Una procedura semplice per acquisire (o “leggere”) dati in ingresso usando tale oggetto prevede di utilizzarlo per creare un oggetto di tipo `Scanner`, in questo modo:

```
new Scanner(System.in)
```

La classe `Scanner` è dotata di un certo numero di metodi, utili per leggere dati dal flusso di ingresso fornito al suo costruttore. Uno di questi è invocato dal seguente programma:

```
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter your age in years: ");
        double age = input.nextDouble(); // età in anni
        System.out.print("Enter your maximum heart rate: ");
        double rate = input.nextDouble(); // frequenza cardiaca massima
        double fb = (rate - age) * 0.65; // frequenza "brucia grassi"
        System.out.println("Your ideal fat-burning heart rate is " + fb);
    }
}
```

Una volta eseguito, il programma visualizzerà nella finestra di console, ad esempio, questo testo:

```
Enter your age in years: 21
Enter your maximum heart rate: 220
Your ideal fat-burning heart rate is 129.35
```

Metodi di `java.util.Scanner`

La classe `Scanner` legge il flusso di input e lo scomponete in *tokens* (“simboli”), che sono stringhe di caratteri separate da specifici *delimitatori*. Un delimitatore è una stringa separatrice appositamente definita, che, in mancanza di istruzioni diverse, è composta da uno o più “caratteri bianchi”, *whitespace*, che sono gli spazi, i caratteri *newline* (“andata a capo”) e *tab* (“tabulazione”). I tokens sono, quindi, separati l’uno dall’altro da una sequenza di caratteri bianchi. Ciascun token può essere letto direttamente come stringa oppure convertito in un valore di un tipo fondamentale dall’oggetto `Scanner`, se ha il formato corretto. In particolare, la classe `Scanner` consente di gestire i tokens mediante i metodi seguenti:

- `hasNext()`: Restituisce `true` se nel flusso di ingresso è presente un altro token.
- `next()`: Restituisce, sotto forma di stringa, il token successivo presente nel flusso di ingresso, generando un errore se non ce ne sono più.

- hasNexttipo():** Restituisce `true` se nel flusso di ingresso è presente un altro token ed è possibile interpretarlo come un valore del *tipo* fondamentale indicato; *tipo* può essere `Boolean`, `Byte`, `Double`, `Float`, `Int`, `Long` o `Short`.
- nexttipo():** Restituisce il token successivo presente nel flusso di ingresso sotto forma di un valore del *tipo* fondamentale indicato, generando un errore se non ci sono più token oppure se il token successivo presente nel flusso non può essere interpretato come un valore del *tipo* fondamentale indicato.

Gli oggetti di tipo `Scanner` possono anche elaborare il flusso d'ingresso riga per riga, ignorando i delimitatori e cercando specifiche espressioni canoniche (*pattern*) mentre procedono, una riga dopo l'altra. Tra i metodi che consentono questo tipo di elaborazione citiamo:

- hasNextLine():** Restituisce `true` se nel flusso di ingresso è presente un'altra riga di testo.
- nextLine():** Acquisisce una riga di testo e la restituisce (senza il `newline` finale). La scansione del flusso proseguirà dopo il `newline` che termina la riga acquisita.
- findInLine(String s):** Cerca nella riga in esame una stringa che corrisponda all'espressione canonica (*regular expression*) *s*. Se c'è, viene restituita e la scansione del flusso procede dal primo carattere che segue la stringa individuata; altrimenti, il metodo restituisce `null` e non fa avanzare il punto di scansione.

Questi metodi possono essere usati assieme ai precedenti, come in questo esempio:

```
Scanner input = new Scanner(System.in);
System.out.print("Please enter an integer: ");
while (!input.hasNextInt()) { // non c'è un numero intero
    input.nextLine();           // "consuma" la riga errata
    System.out.print("Invalid integer; please enter an integer: ");
}
int i = input.nextInt();
```

1.7 Un esempio di programma

In questo paragrafo vedremo un altro esempio di classe Java che illustra molti dei costrutti sintattici definiti nei precedenti paragrafi di questo capitolo. Questa classe `CreditCard` definisce oggetti di tipo “carta di credito” che rappresentano un modello di una versione semplificata delle vere carte di credito, che memorizzano informazioni relative al proprietario (*customer*), alla banca emittente (*issuing bank*), al numero della carta (*account identifier*), al limite di credito (*credit limit*) e al saldo attuale (*current balance*). Non vengono applicati interessi per i rimborsi oltre la scadenza, ma vengono impediti gli utilizzi che porterebbero il saldo della carta al di sopra del limite di credito. La classe è anche dotata del metodo statico `main`, che ne verifica il funzionamento.

Gli elementi principali della definizione della classe `CreditCard` si trovano nel Codice 1.5, mentre il metodo `main` è riportato nel Codice 1.6 e il Codice 1.7 mostra ciò che viene visualizzato dal metodo `main`. Ecco alcune delle caratteristiche principali di questa classe, con riferimento a ciò che illustra.

- La classe definisce cinque variabili di esemplare (righe 3–7), quattro delle quali sono dichiarate `private` e una `protected` (nel prossimo capitolo, parlando di ereditarietà, sfrutteremo il fatto che il membro `balance` sia stato, appunto, definito `protected`).
- La classe definisce due diversi costruttori. La prima versione (che inizia alla riga 9) richiede cinque parametri, tra i quali figura esplicitamente il saldo iniziale del conto. Il secondo costruttore (che inizia alla riga 16) accetta solamente quattro parametri e si avvale dell'uso della speciale parola chiave `this` per invocare la versione con cinque parametri, fornendo un saldo iniziale esplicitamente uguale a zero (un valore ragionevole per la maggior parte dei nuovi conti bancari).
- La classe definisce cinque elementari metodi di accesso (righe 20–24) e due metodi di aggiornamento (`charge` e `makePayment`). Il metodo `charge` sfrutta l'esecuzione condizionale per garantire che un addebito venga rifiutato ogniqualvolta la sua accettazione porterebbe il saldo oltre il limite di credito della carta.
- Alle righe 37–43 è stato inserito un metodo `static` di utilità, di nome `printSummary`.
- Il metodo `main` contiene un array, `wallet`, che memorizza esemplari di `CreditCard`. Tale metodo prevede l'utilizzo di un ciclo `while`, di un ciclo `for` tradizionale e di un ciclo `for-each`, che operano sul contenuto dell'array `wallet`.
- Il metodo `main` illustra la sintassi usata per l'invocazione di metodi tradizionali (cioè non `static`), come `charge`, `getBalance` e `makePayment`, così come quella per l'invocazione di metodi statici (nel caso di `printSummary`).

Codice 1.5: La classe `CreditCard`.

```

1 public class CreditCard {
2     // Variabili di esemplare:
3     private String customer;    // nome del proprietario (es. "John Bowman")
4     private String bank;        // nome della banca (es. "California Savings")
5     private String account;    // numero della carta (es. "5391 0375 9387 5309")
6     private int limit;         // limite di credito (in dollari)
7     protected double balance; // saldo attuale (in dollari)
8
9     // Costruttori:
10    public CreditCard(String cust, String bk, String acnt, int lim, double initBal) {
11        customer = cust;
12        bank = bk;
13        account = acnt;
14        limit = lim;
15        balance = initBal;
16    }
17    public CreditCard(String cust, String bk, String acnt, int lim) {
18        this(cust, bk, acnt, lim, 0.0); // usa saldo zero
19    }
20    // Metodi d'accesso:
21    public String getCustomer() { return customer; }
22    public String getBank() { return bank; }
23    public String getAccount() { return account; }
24    public int getLimit() { return limit; }
25    public double getBalance() { return balance; }
26
27    // Metodi di aggiornamento:
28
29    void charge(int amount) {
30        if (amount > limit - balance) {
31            System.out.println("Saldo insufficiente");
32            return;
33        }
34        balance -= amount;
35    }
36
37    void makePayment(double amount) {
38        if (amount < 0) {
39            System.out.println("Impossibile versare un importo negativo");
40            return;
41        }
42        balance += amount;
43    }
44
45    static void printSummary(CreditCard[] wallet) {
46        int totalBal = 0;
47        for (CreditCard card : wallet) {
48            System.out.println(card.getCustomer() + " " + card.getBank());
49            System.out.println("  Numero: " + card.getAccount());
50            System.out.println("  Limite: " + card.getLimit());
51            System.out.println("  Saldo: " + card.getBalance());
52            totalBal += card.getBalance();
53        }
54        System.out.println("Saldo totale: " + totalBal);
55    }
56}
```

```

26 public boolean charge(double price) { // effettua un addebito
27     if (price + balance > limit)          // se l'addebito fa superare limit
28         return false;                   // rifiuta l'addebito
29     // a questo punto l'addebito è ammmissibile
30     balance += price;                 // aggiorna il saldo
31     return true;                     // comunica la buona notizia
32 }
33 public void makePayment(double amount) { // effettua un rimborso
34     balance -= amount;
35 }
36 // Metodo di utilità: visualizza le informazioni relative a una carta
37 public static void printSummary(CreditCard card) {
38     System.out.println("Customer = " + card.customer);
39     System.out.println("Bank = " + card.bank);
40     System.out.println("Account = " + card.account);
41     System.out.println("Balance = " + card.balance); // cast implicito
42     System.out.println("Limit = " + card.limit);      // cast implicito
43 }
44 // più avanti, il metodo main...
45 }
```

Codice 1.6: Il metodo main della classe CreditCard.

```

1  public static void main(String[] args) {
2      CreditCard[] wallet = new CreditCard[3];
3      wallet[0] = new CreditCard("John Bowman", "California Savings",
4                                  "5391 0375 9387 5309", 5000);
5      wallet[1] = new CreditCard("John Bowman", "California Federal",
6                                  "3485 0399 3395 1954", 3500);
7      wallet[2] = new CreditCard("John Bowman", "California Finance",
8                                  "5391 0375 9387 5309", 2500, 300);
9
10     for (int val = 1; val <= 16; val++) {
11         wallet[0].charge(3*val);
12         wallet[1].charge(2*val);
13         wallet[2].charge(val);
14     }
15
16     for (CreditCard card : wallet) {
17         CreditCard.printSummary(card); // invocazione di metodo statico
18         while (card.getBalance() > 200.0) {
19             card.makePayment(200);
20             System.out.println("New balance = " + card.getBalance());
21         }
22     }
23 }
```

Codice 1.7: Informazioni visualizzate dall'esecuzione del metodo main della classe CreditCard.

```

Customer = John Bowman
Bank = California Savings
Account = 5391 0375 9387 5309
Balance = 408.0
Limit = 5000
New balance = 208.0
New balance = 8.0
Customer = John Bowman
```

```

Bank = California Federal
Account = 3485 0399 3395 1954
Balance = 272.0
Limit = 3500
New balance = 72.0
Customer = John Bowman
Bank = California Finance
Account = 5391 0375 9387 5309
Balance = 436.0
Limit = 2500
New balance = 236.0
New balance = 36.0

```

1.8 Pacchetti e importazione

Il linguaggio Java adotta un approccio utile e generale per l'organizzazione delle classi all'interno dei programmi. Ogni classe pubblica a sé stante che viene definita in Java deve stare in un file distinto, il cui nome deve coincidere con quello della classe, completato con l'estensione `.java`: ad esempio, una classe dichiarata come `public class Window` deve trovarsi nel file `Window.java`. Tale file può contenere anche le definizioni di altre classi, nessuna delle quali, però, può avere visibilità pubblica.

Per agevolare l'organizzazione di grandi archivi di codice, Java consente la creazione di gruppi di definizioni di tipi di dati (come classi e enumerazioni) tra loro correlati, nella forma dei cosiddetti *pacchetti* (*package*). Perché la definizione di un tipo di dato appartenga a un pacchetto di nome *nomePacchetto*, il file contenente il suo codice deve appartenere a una cartella (o *directory*) che abbia lo stesso nome, cioè *nomePacchetto*, e deve iniziare con la riga:

```
package nomePacchetto;
```

Per convenzione, i nomi dei pacchetti devono essere scritti con lettere minuscole. Ad esempio, potremmo definire un pacchetto `architecture`, che definisca classi come `Window` (finestra), `Door` (porta) e `Room` (stanza). Le definizioni di tipi di dati pubblici che si trovino in un file privo di un'esplicita dichiarazione `package` vanno a confluire in quello che viene chiamato *pacchetto standard o di default* (*default package*).

Per fare riferimento, nel codice, a un tipo di dato che si trova all'interno di un pacchetto che non sia quello standard, possiamo usare il suo "nome completo" (*fully qualified name*), basato sulla consueta "notazione punto", con il nome del tipo di dato che costituisce un attributo del nome del pacchetto. Così, ad esempio, potremmo dichiarare una variabile il cui tipo sia `architecture.Window`.

I pacchetti possono, poi, essere ulteriormente organizzati in modo gerarchico all'interno di *sottopacchetti* (*subpackage*). I file delle classi di un sottopacchetto devono trovarsi in una sottocartella della cartella definita per il pacchetto e i nomi completi di tipi definiti nel sottopacchetto sfruttano un ulteriore livello di "notazione punto". Ad esempio, nel pacchetto `java.util` esiste il sottopacchetto `java.util.zip` (che fornisce strumenti per operare con la compressione ZIP) e `java.util.zip.Deflater` è il nome completo della classe `Deflater` definita in tale sottopacchetto.

Tra i molteplici vantaggi derivanti dall'organizzazione delle classi in pacchetti, citiamo il fatto che:

- I pacchetti ci aiutano a evitare i trabocchetti derivanti da nomi che entrano in conflitto. Se tutte le definizioni di tipi si trovassero in un unico pacchetto, potrebbe esistere un'unica classe pubblica di nome `Window`, mentre, usando i pacchetti, possiamo avere una classe `architecture.Window` diversa dalla classe `gui.Window`, usata nell'ambito delle interfacce grafiche per l'interazione con l'utente (`GUI, graphical user interface`).
- È molto più facile rendere disponibile ad altri programmatore un insieme completo di classi, perché le possono riutilizzare, quando queste sono organizzate in un pacchetto.
- Quando alcune definizioni di tipi di dati sono relative a uno scopo ben preciso, se sono raggruppate in un pacchetto, per i programmatore è più semplice ritrovarle all'interno di una grande libreria di classi, comprendendone meglio il loro utilizzo coordinato.
- Le classi che si trovano in uno stesso pacchetto hanno accesso reciproco ai membri che sono definiti con visibilità `public`, `protected` o `default` (cioè qualunque livello di visibilità, purché non `private`).

Enunciati di importazione

Come abbiamo appena detto, possiamo fare riferimento a un tipo di dato definito all'interno di un pacchetto usando il suo nome completo: ad esempio, la classe `Scanner`, vista nel Paragrafo 1.6, è definita nel pacchetto `java.util`, per cui la potremmo utilizzare scrivendo `java.util.Scanner` e potremmo, in un nostro progetto, dichiarare e costruire un nuovo esemplare di tale classe usando un enunciato come questo:

```
java.util.Scanner input = new java.util.Scanner(System.in);
```

Risulta però subito evidente che scrivere nomi così lunghi per i tipi di dati ogni volta che si fa riferimento a una classe esterna al pacchetto in cui si sta lavorando può essere molto noioso. Per questo motivo, in Java è possibile utilizzare la parola chiave `import` per includere nel file su cui si sta lavorando classi di altri pacchetti o anche interi pacchetti, con tutte le loro classi. Per importare una singola classe definita in uno specifico pacchetto scriviamo, all'inizio del file, la riga seguente:

```
import nomePacchetto.nomeClasse;
```

Nel Paragrafo 1.6, ad esempio, abbiamo importato la classe `Scanner` del pacchetto `java.util` scrivendo così:

```
import java.util.Scanner;
```

dopodiché, nel file che contiene tale direttiva, abbiamo potuto usare una sintassi decisamente meno pesante:

```
Scanner input = new Scanner(System.in);
```

Occorre, però, ricordare che non è ammessa l'importazione di una classe, usando la direttiva `import`, se un'altra classe con lo stesso nome è presente nel file che si sta scrivendo oppure vi

è stata importata da un altro pacchetto. Ad esempio, non potremmo importare entrambe le classi `architecture.Window` e `gui.Window`, per poi usare il nome incompleto `Window`, perché, evidentemente, sarebbe ambiguo.

Importare un intero pacchetto

Se sappiamo che useremo molte classi definite in uno stesso pacchetto, possiamo importarle tutte usando un asterisco (*) come *carattere jolly*, in questo modo:

```
import nomePacchetto.*;
```

Se un nome definito nel file entra in conflitto con uno presente nel pacchetto così importato, il nome locale può continuare a essere utilizzato con il suo nome incompleto, mentre per usare la omonima classe del pacchetto importato bisognerà comunque utilizzare il suo nome completo. Se, invece, si verifica un conflitto tra i nomi di classi definite in due diversi pacchetti importati in questo modo, *nessuna* delle due potrà essere usata con il nome incompleto. Se, ad esempio, importiamo i due ipotetici pacchetti `architecture` e `gui`:

```
import architecture.*; // immaginiamo che contenga una classe Window
import gui.*;           // immaginiamo che contenga una classe Window
```

allora nel nostro programma dobbiamo usare i nomi completi, `architecture.Window` oppure `gui.Window`.

1.9 Sviluppo del software

Lo sviluppo tradizionale del software prevede diverse fasi, le principali delle quali sono:

1. Progettazione
2. Scrittura del codice
3. Collaudo e *debugging*, cioè eliminazione degli errori individuati con il collaudo

In questo paragrafo discuteremo brevemente il ruolo di ciascuna di queste fasi e presenteremo alcune valide strategie di programmazione in Java, tra le quali l'adesione a stili predeterminati per la scrittura del codice e per l'assegnazione dei nomi agli identificatori, la documentazione con un formato ben preciso e il collaudo.

1.9.1 Progettazione

Nella programmazione orientata agli oggetti, la fase di progettazione è forse la più importante dell'intero processo di sviluppo del software. È nella fase di progettazione che si decide come suddividere in più classi il lavoro che deve essere svolto dal programma, che si decide quali dovranno essere le interazioni tra le classi così individuate, quali dati dovranno essere memorizzati in ogni esemplare di ciascuna classe e quali azioni questi esemplari dovranno poter compiere. In effetti, il problema più complesso per i programmatore principianti è decidere quali classi sia opportuno definire per realizzare un determinato programma. No-

nonostante sia difficile poter giungere a regole ben precise, esistono alcune "buone pratiche" che si possono solitamente applicare nel momento in cui si debba, appunto, decidere quali classi definire:

- **Responsabilità.** Suddividere il lavoro da svolgere tra diversi *attori*, ciascuno con una propria e diversa responsabilità. Cercare di descrivere le responsabilità usando verbi. Questi attori saranno le classi del programma.
- **Indipendenza.** Definire il lavoro che deve essere svolto da ciascuna classe in modo che sia quanto più indipendente possibile da quello svolto da altre. Suddividere le responsabilità tra le classi in modo che ciascuna di esse sia autonoma in relazione ad alcuni aspetti del programma. Definire ciascun dato (sotto forma di variabile di esemplare) all'interno di quella classe che ha la competenza di mettere in atto azioni che richiedono l'accesso a quello specifico dato.
- **Comportamenti.** Definire i comportamenti caratteristici di ciascuna classe con attenzione e precisione, in modo da comprendere bene, per ciascuna azione portata a termine da una classe, le conseguenze nei confronti delle altre classi che interagiscono con essa. Tali comportamenti definiranno i metodi della classe, e l'insieme dei suoi comportamenti definisce il *protocollo* con cui altre sezioni di codice possono interagire con oggetti che siano esemplari della classe.

La definizione delle classi, con le loro variabili di esemplare e i loro metodi, è una fase cruciale nel progetto di un programma orientato agli oggetti. Con il passare del tempo, un buon programmatore aumenterà la propria capacità di portare a termine questa fase, perché con l'esperienza maturata sarà sempre più facile notare schemi ricorrenti nei requisiti di un programma, da mettere in relazione con quanto visto in precedenza.

Uno strumento piuttosto diffuso per lo sviluppo iniziale ad alto livello di un progetto consiste nell'utilizzo delle cosiddette *schede CRC*, dove CRC sta per Class-Responsibility-Collaborator (cioè Classe-Responsabilità-Collaboratore): sono semplicemente schede che aiutano a suddividere i compiti che devono essere portati a termine da un programma. L'idea su cui si basa questo strumento è quella di avere una scheda per rappresentare ciascun componente del progetto: ogni componente, poi, diventerà una classe del programma. Nella parte alta di ogni scheda scriviamo il nome del componente che rappresenta; nella parte sinistra della scheda iniziamo a scrivere le responsabilità del componente, mentre nella parte destra elenchiamo i collaboratori del componente, cioè gli altri componenti con cui il componente dovrà interagire per svolgere i propri compiti.

La progettazione procede ripetutamente attraverso cicli di tipo azione/attore, nei quali per prima cosa viene identificata un'azione (cioè una responsabilità), poi viene individuato un attore (cioè un componente) che la possa portare a termine nel migliore dei modi. Il progetto è completo quando tutte le azioni sono state assegnate a un attore. Usando delle piccole schede durante l'intero processo (piuttosto che fogli di carta di maggiori dimensioni) confidiamo nel fatto che ciascun componente debba avere un ristretto insieme di responsabilità e di collaboratori: costringendoci al rispetto di questa regola empirica, sarà più facile giungere, alla fine, a classi gestibili.

Mentre il progetto prende forma, possiamo usare schemi UML (Unified Modeling Language) per delineare l'organizzazione del programma, seguendo un approccio standard

per illustrare e documentare il progetto. Gli schemi UML usano una struttura grafica standard per rappresentare progetti software orientati agli oggetti e sono disponibili molti strumenti di ausilio per disegnarli al computer. Uno dei possibili schemi UML prende il nome di *schemma di classe* (*class diagram*).

La Figura 1.5 riporta un esempio di schema di classe, corrispondente alla nostra classe *CreditCard* progettata nel Paragrafo 1.7. Lo schema è costituito da tre sezioni: la prima riporta il nome della classe, la seconda individua un insieme plausibile di variabili di esemplare, e la terza delinea i metodi di cui si suggerisce la realizzazione. La dichiarazione dei tipi di dati coinvolti (variabili, parametri e valori restituiti dai metodi) avviene nelle posizioni più opportune, dopo un carattere “due punti”, mentre la visibilità di ciascun membro (variabile o metodo) è indicata alla sinistra del suo nome, dove un carattere ‘+’ sta per **public**, ‘#’ per **protected** e ‘-’ per **private**.

CreditCard	
classe:	
variabili:	<ul style="list-style-type: none"> - customer : String - bank : String - account : String <ul style="list-style-type: none"> - limit : int # balance : double
metodi:	<ul style="list-style-type: none"> + getCustomer() : String + getBank() : String + charge(price : double) : boolean + makePayment(amount : double) <ul style="list-style-type: none"> + getAccount() : String + getLimit() : int + getBalance() : double

Figura 1.5: Lo schema di classe UML per la classe *CreditCard* del Paragrafo 1.7.

1.9.2 Pseudocodice

Come passo intermedio verso l’implementazione di un progetto, spesso si chiede ai programmatore di descrivere gli algoritmi in un linguaggio destinato all’analisi umana, chiamato *pseudocodice*: non si tratta di un linguaggio di programmazione per calcolatori, ma è comunque più strutturato della normale prosa con cui si scrive in un linguaggio naturale. È, appunto, un insieme di parole in linguaggio naturale e di costrutti sintattici tipici della programmazione di alto livello, utile per descrivere le idee principali su cui si basa l’implementazione di una struttura dati o di un algoritmo. Dal momento che lo pseudocodice è destinato a lettori umani, e non a calcolatori, possiamo usarlo per comunicare idee ad alto livello, senza appesantirlo con i dettagli di basso livello relativi all’implementazione. Al tempo stesso, dovremmo cercare di non trascurare le fasi principali: come in molte altre forme di comunicazione umana, il fatto di saper trovare il giusto compromesso è un’abilità molto importante, che si affina con il tempo e con l’esercizio.

In verità, non esiste una definizione precisa di “pseudocodice” come linguaggio. Nonostante questo, per cercare di essere chiari, diciamo che lo pseudocodice contiene parole di un linguaggio naturale e costrutti sintattici standard presi dai più diffusi e moderni linguaggi di programmazione, come C, C++ e Java, tra i quali troviamo:

- *Espressioni*. Per scrivere espressioni numeriche e booleane, usiamo i normali simboli matematici. Per coerenza con Java, usiamo il segno “=” come operatore di assegnazione.

zione e il simbolo “==” per esprimere la relazione di uguaglianza nelle espressioni booleane.

- **Dichiarazioni di metodi.** Per dichiarare il nuovo metodo *nome* e i suoi parametri scriviamo **Algorithm nome(param1, param2, ...)**.
- **Strutture decisionali.** *if condizione then azioniSeVera [else azioniSeFalsa]*. Usiamo il rientro verso destra (o *indentazione*) per raggruppare le azioni che fanno parte della sezione *azioniSeVera* o *azioniSeFalsa*.
- **Cicli while.** *while condizione do azioni*. Usiamo l'*indentazione* per raggruppare le *azioni*.
- **Cicli repeat.** *repeat azioni until condizione*. Usiamo l'*indentazione* per raggruppare le *azioni*.
- **Cicli for.** *for variabile – incremento – definizione do azioni*. Usiamo l'*indentazione* per raggruppare le *azioni*.
- **Individuazione di array.** *A[i]* rappresenta la *i*-esima cella dell'array *A*. Le celle di un array *A* di dimensione *n* hanno indici che vanno da 0 a *n* – 1 (come in Java).
- **Invocazioni di metodi.** *object.method(args)*; l'indicazione esplicita di *object* può essere omessa se è chiara dal contesto.
- **Terminazione di metodi.** *return valore*. Questa operazione restituisce il *valore* indicato al metodo invocante.
- **Commenti.** { questo è un commento }. Racchiudiamo i commenti tra una coppia di parentesi graffe.

1.9.3 Scrittura del codice

Un'altra fase chiave nell'implementazione di un programma orientato agli oggetti è la scrittura del codice che descrive le classi e i loro dati e metodi. Allo scopo di favorire la velocità di apprendimento di questa abilità, presenteremo in vari punti del libro *schemi progettuali (design pattern)* ricorrenti per la realizzazione, appunto, di programmi orientati agli oggetti (Paragrafo 2.1.3). Questi schemi sono utili come base per la definizione di classi e delle loro interazioni con altre.

Dopo aver deciso quali saranno le classi del nostro programma e le loro responsabilità, eventualmente delineando i loro comportamenti in pseudocodice, siamo pronti per iniziare effettivamente la scrittura del codice al computer. Possiamo scrivere il codice sorgente Java delle classi del nostro programma usando un *editor* di testo a sé stante (come *emacs*, *Wordpad* o *vi*) oppure interno a un *ambiente di sviluppo integrato* o IDE (*integrated development environment*), come *Eclipse*.

Dopo aver scritto il codice di una classe (o di un pacchetto), lo compiliamo, invocando un compilatore Java. Se non stiamo usando un IDE, effettuiamo la compilazione invocando un programma, come *javac*, fornendo il nostro file sorgente come argomento. Se, invece, stiamo utilizzando un IDE, compiliamo il nostro programma selezionando la sua opzione appropriata, probabilmente usando il mouse. Se siamo fortunati e il nostro programma non contiene errori, questo processo di compilazione crea dei file il cui nome è caratterizzato dall'estensione *.class*.

Al contrario, se il programma contiene degli errori di sintassi, questi vengono identificati e dobbiamo tornare all'*editor* per correggere le righe di codice sbagliate. Una volta che abbiamo eliminato tutti gli errori di sintassi e creato il codice compilato corrispondente al nostro codice sorgente, possiamo eseguire il nostro programma invocando un comando,

come `java` (al di fuori di un IDE), oppure selezionando il comando dell'IDE dedicato a questo, che probabilmente si chiama “run” o “esegui”. Quando un programma Java viene eseguito in questo modo, l'ambiente di esecuzione individua la cartella contenente la classe da eseguire e le altre classi da essa utilizzate (e così via) sulla base di una specifica variabile di ambiente del sistema operativo, che prende il nome di `CLASSPATH`. Questa variabile definisce un elenco ordinato di cartelle in cui cercare, i cui nomi sono separati da caratteri “due punti” in Unix/Linux e da “punto e virgola” in DOS/Windows. Ecco un esempio dell'assegnazione di un valore a `CLASSPATH` nel sistema operativo DOS/Windows:

```
SET CLASSPATH=.;C:\java;C:\Program Files\Java\
```

mentre un esempio in Unix/Linux potrebbe essere questo:

```
setenv CLASSPATH ".:/usr/local/java/lib:/usr/netscape/classes"
```

In entrambi i casi, il “punto” iniziale nell'elenco si riferisce alla cartella in cui viene invocato l'ambiente di esecuzione.

1.9.4 Documentazione e stile

Javadoc

Per incoraggiare il corretto utilizzo dei commenti a blocchi e la generazione automatica della documentazione, l'ambiente di programmazione Java contiene un programma, `javadoc`, che serve, appunto, a generare automaticamente la documentazione delle classi e dei pacchetti. Questo programma analizza una raccolta di file sorgenti Java che siano stati commentati usando ben determinate parole chiave, chiamate *marcatori* (*tag*), e genera una serie di documenti HTML che descrivono le classi contenute in quei file, con i loro metodi, variabili e costanti. A titolo di esempio, la Figura 1.6 mostra una parte della documentazione generata per la nostra classe `CreditCard`.

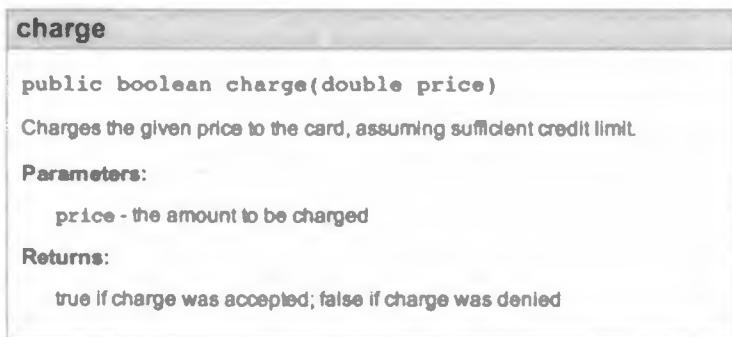


Figura 1.6: Documentazione prodotta da javadoc per il metodo `CreditCard.charge`.

Ogni commento destinato a javadoc è un commento a blocchi che inizia con “`/**`” e termina con “`*/`” (ogni riga fra quella iniziale e quella finale può cominciare con un asterisco, “`*`”, che viene ignorato. Si suppone che il commento inizi con una frase descrittiva, seguita da righe

con un formato speciale, che iniziano con marcatori opportuni. Un commento a blocchi che preceda la definizione di una classe, la dichiarazione di una variabile di esemplare o la definizione di un metodo viene elaborato da javadoc per descrivere quello stesso elemento (classe, variabile o metodo). I marcatori principali sono:

- **@author testo**: identifica gli autori della classe (un marcatore per autore, uno per riga).
- **@throws nomeEccezione descrizione**: descrive una condizione di errore che viene segnalata da questo metodo (Paragrafo 2.4).
- **@param nomeParametro descrizione**: descrive un parametro accettato da questo metodo.
- **@return descrizione**: descrive il valore restituito (tipo e intervalli di valori validi) da questo metodo.

Ci sono anche altri marcatori: il lettore interessato ad approfondire l'argomento è invitato a consultare la documentazione di javadoc. Per motivi di spazio, non sempre aggiungiamo ai programmi presentati in questo libro i commenti secondo lo stile di javadoc, ma ne presentiamo un esempio nel Codice 1.8.

Codice 1.8: Una parte della definizione della classe CreditCard, già presentata nel Codice 1.5, qui con i commenti previsti dallo stile di javadoc.

```
1  /**
2  * Un semplice modello di carta di credito.
3  *
4  * @author Michael T. Goodrich
5  * @author Roberto Tamassia
6  * @author Michael H. Goldwasser
7  */
8 public class CreditCard {
9 /**
10 * Costruisce un nuovo esemplare di carta di credito.
11 * @param cust il nome del proprietario (es. "John Bowman")
12 * @param bk   il nome della banca (es. "California Savings")
13 * @param acnt il numero della carta (es. "5391 0375 9387 5309")
14 * @param lim  il limite di credito (in dollari)
15 * @param initBal il saldo attuale (in dollari)
16 */
17 public CreditCard(String cust, String bk, String acnt, int lim, double initBal) {
18     customer = cust;
19     bank = bk;
20     account = acnt;
21     limit = lim;
22     balance = initBal;
23 }
24 /**
25 * Addebita sulla carta il prezzo indicato, se non si supera il limite di credito.
26 * @param price la somma da addebitare
27 * @return true se l'addebito viene accettato, altrimenti false
28 */
29 public boolean charge(double price) { // effettua un addebito
30     if (price + balance > limit)          // se l'addebito fa superare limit
31         return false;                    // rifiuta l'addebito
32     // a questo punto l'addebito è ammissibile
33 }
```

```

    balance += price;           // aggiorna il saldo
    return true;                // comunica la buona notizia
}

/**
 * Elabora un rimborso che riduce il saldo.
 * @param amount la somma rimborsata
 */
public void makePayment(double amount) { // effettua un rimborso
    balance -= amount;
}
// la classe proseguirebbe...

```

Leggibilità e convenzioni stilistiche

I programmi dovrebbero essere di facile lettura e comprensione, per cui un buon programmatore dovrebbe essere molto attento allo stile adottato nella scrittura del codice, sviluppando metodologie che comunichino gli aspetti più rilevanti di un programma tanto a lettori umani quanto al calcolatore. Molto è stato detto in merito alla scrittura di codice secondo un “buono stile di programmazione” e alcune delle linee guida sono qui riassunte:

- Usare nomi significativi per gli identificatori. Cercare di scegliere nomi che si possono leggere a voce alta e che riflettano l’azione, la responsabilità o il dato che sono chiamati a identificare. In Java, è ormai tradizione consolidata usare identificatori con la prima lettera maiuscola, tranne nel caso di nomi di variabili e di metodi: secondo tale convenzione, `Date`, `Vector` e `DeviceManager` identificheranno classi, mentre `isFull()` e `insertItem()` sono nomi di metodi e `studentName` e `studentHeight` sono nomi di variabili.
- Al posto di valori letterali, usare costanti dotate di nome oppure tipi enumerativi. L’inserimento di una serie di definizioni di valori costanti all’interno di una classe migliora la sua leggibilità, robustezza e modificabilità: si potranno usare i nomi di tali costanti all’interno della classe stessa o in altre che abbiano bisogno di fare riferimento ai valori speciali usati in quella classe. La tradizione vuole che, in Java, i nomi delle costanti siano scritti in maiuscolo, in questo modo:

```

public class Student {
    public static final int MIN_CREDITS = 12;
    public static final int MAX_CREDITS = 24;
    public enum Year {FRESHMAN, SOPHMORE, JUNIOR, SENIOR};
    // seguono variabili di esemplare, costruttori, metodi...
}

```

- Usare il rientro verso destra (o indentazione) per evidenziare i blocchi di enunciati. Solitamente i programmati usano un rientro di 4 spazi, ma in questo libro ne usiamo soltanto 2, per evitare che il codice finisca oltre il margine destro della pagina.
- Organizzare ciascuna classe in questo modo:
 1. Costanti
 2. Variabili di esemplare
 3. Costruttori
 4. Metodi

Alcuni programmatori, in Java, preferiscono mettere le definizioni delle variabili di esemplare alla fine della classe, ma noi preferiamo inserirle all'inizio, perché riteniamo che questo agevoli la lettura del codice e la comprensione dei dati su cui i metodi operano.

- Scrivere commenti che aggiungano significato al programma e chiariscano eventuali ambiguità o costrutti un po' contorti. I commenti a fine riga sono utili per spiegazioni molto brevi e non dovrebbero andare oltre una singola frase. I commenti a blocchi sono, invece, adatti per spiegare lo scopo di un metodo o per aiutare a comprendere parti di codice un po' complicate.

1.9.5 Collaudo e debugging

Si chiama collaudo o *testing* il processo che verifica in modo sperimentale la correttezza di un programma, mentre la fase di *debugging* (o eliminazione dei *bug*, cioè degli errori nel software) prevede l'esecuzione passo dopo passo di un programma, per scoprire gli errori che vi sono contenuti. Spesso il collaudo e il debugging sono le fasi di programmazione che richiedono più tempo all'interno dell'intero processo di sviluppo di un programma.

Collaudo

Un attento piano di collaudo è parte integrante della scrittura di un programma. Dato che spesso non è possibile, dal punto di vista pratico, verificare la correttezza di un programma applicando in ingresso tutti i valori possibili dei dati, bisogna cercare di eseguire il programma usando un sottoinsieme di tali casi che sia rappresentativo. Come minimo, dobbiamo accertarci che ciascun metodo del programma venga collaudato almeno una volta (a questo riguardo, si parla di "copertura dei metodi", *method coverage*). Ancor meglio è garantire che ciascun enunciato del codice del programma venga eseguito almeno una volta ("copertura degli enunciati", *statement coverage*).

Spesso i programmi tendono a sbagliare in presenza di *casi speciali* dei dati in ingresso, quindi tali casi vanno identificati e collaudati con la massima attenzione. Ad esempio, quando si collauda un metodo che ordina un array di numeri interi (cioè dispone i valori all'interno dell'array in modo che crescano al crescere dell'indice della cella), bisogna prendere in esame i seguenti casi speciali dei dati in ingresso:

- L'array ha lunghezza zero (cioè non ha elementi).
- L'array ha un solo elemento.
- Gli elementi dell'array sono tutti uguali tra loro.
- L'array è già ordinato.
- L'array è ordinato in senso inverso.

Oltre ai casi speciali per i dati in ingresso al programma, bisogna anche considerare le situazioni speciali in cui si possono trovare le strutture usate dal programma stesso. Ad esempio, se viene usato un array per memorizzare dati, dobbiamo essere sicuri che i casi limite (*boundary case*), come l'inserimento o la rimozione di un dato all'inizio o alla fine del sottoarray che contiene effettivamente dati vengano gestiti nel modo corretto.

Nonostante sia essenziale utilizzare casi di prova progettati a mano, spesso è altrettanto utile eseguire il programma su grandi raccolte di dati in ingresso generati a caso: la classe *Random* del pacchetto *java.util* consente di generare numeri pseudocasuali.

Tra le classi e i metodi di un programma esiste una gerarchia, determinata dalla relazione invocante-invocato. In particolare, un metodo *A* sta sopra (nella gerarchia) al metodo *B* se *A* invoca *B*. In relazione a questo, esistono due strategie principali per il collaudo, che differiscono per l'ordine in cui i metodi vengono collaudati: collaudo *top-down* (cioè dall'alto verso il basso) e collaudo *bottom-up* (dal basso verso l'alto).

Il collaudo top-down procede dall'alto in basso nella gerarchia dei metodi del programma e tipicamente viene utilizzato assieme alla strategia dello *studding*, cioè sostituendo i metodi dei livelli inferiori della gerarchia con degli *stub* ("adattatori"), metodi semplificati che ne simulano la funzionalità. Ad esempio, se il metodo *A* invoca il metodo *B* per acquisire la prima riga di un file di testo, quando si collauda *A* si può sostituire *B* con uno stub che restituisca una stringa prefissata.

Il collaudo bottom-up, al contrario, procede dai metodi di più basso livello a quelli di livello più elevato. Per prima cosa vengono collaudati i metodi del livello più basso, che non invocano nessun altro metodo, poi si passa a collaudare i metodi che invocano soltanto metodi del livello più basso, già collaudati, e così via. Analogamente, una classe che non dipenda da nessun'altra classe può essere collaudata prima di altre classi, che magari da essa dipendono. Questa forma di collaudo viene solitamente chiamata *collaudo di unità* (*unit testing*), perché le funzionalità di uno specifico componente di un progetto software, che magari è di grandi dimensioni, vengono collaudate tenendolo isolato dagli altri componenti. Se usata correttamente, questa strategia aiuta a isolare la causa di eventuali errori, che vengono imputati al componente in fase di collaudo, dal momento che i componenti di livello inferiore che vengono utilizzati durante l'esecuzione dovrebbero essere già stati collaudati in modo approfondito.

Java consente di eseguire collaudi automatici a vari livelli. Abbiamo già visto come al metodo statico `main` di una classe possa essere assegnato il compito di eseguire collaudi della funzionalità della classe stessa (come abbiamo fatto per la classe `CreditCard` nel Codice 1.6). Questi collaudi vengono eseguiti invocando la macchina virtuale Java direttamente sulla classe, invece che sulla classe principale dell'intera applicazione. Quando, invece, Java mette in esecuzione la classe principale, il codice di questi metodi `main` "secondari" viene ignorato.

Un ausilio più rilevante per l'automazione del collaudo di unità viene fornito dall'ambiente `JUnit`, che non fa parte dello strumento di sviluppo standard di Java ma è disponibile gratuitamente e liberamente all'indirizzo www.junit.org. Questo *framework* consente di raggruppare singoli casi di prova in *test suite* di più grandi dimensioni, per poi eseguire tali insiemi di prove, analizzandone i risultati. Nella fase di manutenzione del software, poi, è opportuno eseguire il cosiddetto *collaudo regressivo* (*regression testing*), nel quale viene di nuovo utilizzata l'automazione per rieseguire tutti i casi di prova, per garantire che le modifiche apportate al software non abbiano introdotto nuovi errori in componenti già collaudati in precedenza.

Debugging

La tecnica di debugging più semplice consiste nell'utilizzo di *enunciati di visualizzazione* che tengano traccia dei valori delle variabili durante l'esecuzione del programma. Il problema derivante da questo approccio è che prima o poi tali enunciati dovranno essere eliminati o commentati, in modo che non vengano eseguiti quando il programma assume la sua forma definitiva e diventa operativo.

Un approccio migliore prevede di eseguire il programma mediante un *debugger*, che è un ambiente di esecuzione speciale, dedicato al controllo e al monitoraggio dell'esecuzione di un programma. La funzionalità di base messa a disposizione da un debugger è l'inserimento di *punti di interruzione* (*breakpoint*) all'interno del codice. Quando il programma viene eseguito con il debugger, si blocca al raggiungimento di uno dei punti di interruzione e, mentre l'esecuzione è bloccata, è possibile ispezionare il valore delle variabili in quel momento. Oltre all'utilizzo di punti di interruzione fissi, i debugger più evoluti consentono di specificare *breakpoint condizionali*, che vengono abilitati soltanto nel momento in cui viene soddisfatta una data espressione.

Lo strumento di sviluppo standard di Java contiene un debugger elementare, *jdb*, dotato di un'interfaccia a riga di comando, non grafica, ma la maggior parte degli IDE per la programmazione in Java mettono a disposizione ambienti di debugging grafici e molto evoluti.

1.10 Esercizi

Riepilogo e approfondimento

- R-1.1 Scrivere un breve metodo Java, `inputAllBaseTypes`, che acquisisca dal dispositivo di input standard un valore per ciascuno dei tipi di dati fondamentali, visualizzandolo poi sul dispositivo di output standard.
- R-1.2 Immaginare di aver creato un array *A* di oggetti di tipo `GameEntry` (una classe che ha un campo intero, `score`) e di aver clonato *A*, memorizzando il risultato nell'array *B*. Se, dopo di ciò, si assegna il valore 550 a *A[4].score*, quale sarà il valore di `score` dell'oggetto `GameEntry` a cui fa riferimento *B[4]*?
- R-1.3 Scrivere un breve metodo Java, `isMultiple`, che ha come parametri due valori di tipo `long`, *n* e *m*, e restituisce `true` se e solo se *n* è un multiplo di *m*, cioè se esiste un numero intero *i* tale che *n = mi*.
- R-1.4 Scrivere un breve metodo Java, `isEven`, che ha un parametro di tipo `int`, *i*, e restituisce `true` se e solo se *i* è pari. Il metodo non può usare operatori di moltiplicazione, divisione o resto della divisione intera.
- R-1.5 Scrivere un breve metodo Java che riceve un numero intero *n* e restituisce la somma di tutti i numeri interi positivi minori di o uguali a *n*.
- R-1.6 Scrivere un breve metodo Java che riceve un numero intero *n* e restituisce la somma di tutti i numeri interi positivi dispari minori di o uguali a *n*.
- R-1.7 Scrivere un breve metodo Java che riceve un numero intero *n* e restituisce la somma dei quadrati di tutti i numeri interi positivi minori di o uguali a *n*.
- R-1.8 Scrivere un breve metodo Java che conta e restituisce il numero di vocali presenti in una data stringa di caratteri.
- R-1.9 Scrivere un breve metodo Java che usa un esemplare di `StringBuilder` per eliminare tutti i segni di punteggiatura da una stringa *s* che memorizza una frase, trasformando, ad esempio, la stringa "Let's try, Mike!" nella stringa "Lets try Mike".
- R-1.10 Scrivere una classe Java, `Flower`, che abbia tre variabili di esemplare di tipo `String`, `int` e `float`, che, rispettivamente, rappresentino il nome di un fiore, il numero dei suoi petali e il suo prezzo. La classe deve avere un costruttore che inizializza ciascuna

variabile a un valore appropriato, e metodi che consentano di assegnare un valore a ciascuna singola variabile, nonché di ispezionarla.

- R-1.11 Modificare la classe `CreditCard` definita nel Codice 1.5 in modo che disponga di un metodo che consenta di modificare il limite di credito.
- R-1.12 Modificare la classe `CreditCard` definita nel Codice 1.5 in modo che ignori qualunque richiesta di elaborazione di un rimborso con valore negativo.
- R-1.13 Modificare la dichiarazione del primo ciclo `for` nel metodo `main` del Codice 1.6 in modo che gli addebiti siano tali che soltanto una delle tre carte di credito tenti di superare il proprio limite di credito. Di quale carta si tratterà?

Creatività

- C-1.14 Scrivere in pseudocodice la descrizione di un metodo che inverte il contenuto di un array di n numeri interi, in modo che i numeri siano, alla fine, elencati in ordine opposto a quello in cui si trovavano all'inizio. Poi, confrontare questo metodo con l'equivalente metodo della libreria Java che risolve lo stesso problema.
- C-1.15 Scrivere in pseudocodice la descrizione di un metodo che trova il valore minimo e massimo in un array di numeri interi. Poi, confrontare questo metodo con l'equivalente metodo della libreria Java che risolve lo stesso problema.
- C-1.16 Scrivere un breve programma che acquisisce in ingresso tre numeri interi, a , b e c , usando la finestra di console di Java, per poi determinare se possano essere usati, in ordine, in una delle seguenti formule, rendendola corretta: " $a + b = c$ ", " $a = b - c$ ", " $a \cdot b = c$ ".
- C-1.17 Scrivere un breve metodo Java che riceve come parametro un array di valori di tipo `int` e determina se esiste una coppia di elementi distinti dell'array il cui prodotto sia un numero pari.
- C-1.18 La *norma di ordine p* di un vettore $v = (v_1, v_2, \dots, v_n)$ in uno spazio n -dimensionale è così definita:

$$\|v\| = \sqrt[p]{v_1^p + v_2^p + \dots + v_n^p}$$

Nel caso speciale $p = 2$, il risultato è la tradizionale *norma euclidea*, che rappresenta la lunghezza del vettore. Ad esempio, 5 è la norma euclidea di un vettore bidimensionale avente coordinate (4, 3). Implementare un metodo, `norm`, in modo tale che `norm(v, p)` restituisca la *norma di ordine p* di v e `norm(v)` restituisca la norma euclidea di v , con v rappresentato come un array di coordinate.

- C-1.19 Scrivere un programma Java che riceve come valore in ingresso un numero intero positivo maggiore di 2 e visualizza per quante volte tale numero può essere ripetutamente diviso per 2 prima di ottenere un valore minore di 2.
- C-1.20 Scrivere un programma Java che riceve in ingresso un array di valori di tipo `float` e determina se tutti i numeri sono tra loro diversi (cioè se sono tutti distinti).
- C-1.21 Scrivere un metodo Java che riceve come parametro un array contenente l'insieme di tutti i numeri interi compresi tra 1 e 52 e li mescola in ordine casuale, visualizzando il risultato. Il metodo deve produrre uno qualunque dei possibili risultati, ciascuno con la medesima probabilità.
- C-1.22 Scrivere un breve programma Java che visualizza tutte le possibili stringhe composte usando i caratteri 'c', 'a', 't', 'd', 'o', e 'g', ciascuno una e una sola volta.

- C-1.23** Scrivere un breve programma Java che riceve in ingresso due array, *a* e *b*, di lunghezza *n*, contenenti valori di tipo `int`, e restituisce il prodotto di *a* per *b*, definito in modo che il risultato sia un array *c* di lunghezza *n* tale che $c[i] = a[i] \cdot b[i]$ con $i = 0, \dots, n - 1$.
- C-1.24** Modificare la classe `CreditCard` vista nel Codice 1.5 in modo che il metodo `printSummary` diventi *non statico*, modificando poi di conseguenza il metodo `main` del Codice 1.6.
- C-1.25** Modificare la classe `CreditCard` aggiungendovi il metodo `toString()` che restituisca, sotto forma di oggetto di tipo `String`, una rappresentazione della carta di credito (invece di visualizzare tali informazioni, come fa il metodo `printSummary`). Modificare, poi, il metodo `main` del Codice 1.6 in modo che usi il normale metodo `println`.

Progettazione

- P-1.26** Scrivere un breve programma Java che acquisisce tutte le righe fornite dall'utente tramite il dispositivo di input standard e le visualizza sul dispositivo di output standard in ordine inverso rispetto a quello in cui le ha ricevute (ogni riga è esattamente uguale a una delle righe ricevute in ingresso, ma l'ordine in cui le righe vengono visualizzate è invertito rispetto ai dati in ingresso).
- P-1.27** Scrivere un programma Java che sia in grado di simulare il funzionamento di una semplice calcolatrice, usando la finestra di console di Java e i dispositivi di input e output standard. Ogni informazione fornita in input alla calcolatrice, sia un numero, come 12.34 o 1034, o un operatore, come + o =, si trova su una riga a sé stante. Dopo l'acquisizione di ogni riga, il programma deve visualizzare ciò che verrebbe prodotto da una calcolatrice.
- P-1.28** Un tipico compito di punizione per gli scolari prevede di scrivere ripetutamente una stessa frase. Scrivere un programma Java che visualizza cento volte la frase "I will never spam my friends again" (*non invierò mai più messaggi indesiderati ai miei amici*), numerando ciascuna frase visualizzata e compiendo otto diversi errori di battitura casuali.
- P-1.29** Il *paradosso dei compleanni* afferma che la probabilità che almeno due persone in una stanza siano nate nello stesso giorno dell'anno è maggiore del 50% se le persone presenti, *n*, sono più di 23. Questa proprietà, in effetti, non è un paradosso, ma molte persone rimangono sorprese da questa affermazione. Progettare un programma Java che verifichi la correttezza del paradosso, facendo una serie di esperimenti di generazione casuale di date dei compleanni, con $n = 5, 10, 15, 20, \dots, 100$.
- P-1.30** (*Per chi conosce i metodi delle interfacce grafiche in Java*). Definire una classe, `GraphicalTest`, che consenta di verificare la funzionalità della classe `CreditCard` vista nel Codice 1.5 usando pulsanti e campi di testo.

Note

Per avere informazioni più dettagliate sul linguaggio di programmazione Java, rimandiamo il lettore al sito web di Java (<http://www.java.com>), oltre che ad alcuni ottimi libri, tra i quali vogliamo citare quelli di Arnold, Gosling e Holmes [8], di Flanagan [33] e di Horstmann e Cornell [47, 48].

2

Progettazione orientata agli oggetti

2.1 Obiettivi, principi e schemi ricorrenti

Come suggerito dal nome, nel paradigma orientato agli oggetti gli “attori” principali vengono chiamati *oggetti*, e ciascun oggetto è un *esemplare* o istanza di una *classe*. Ogni classe presenta al mondo esterno una vista sintetica e coerente degli oggetti che ne sono esemplari, senza esporre dettagli non necessari e senza fornire ad altri l’accesso al funzionamento interno degli oggetti. Tipicamente la definizione di una classe specifica i *campi di dati*, detti anche *variabili di esemplare*, che sono contenuti in un oggetto, così come i *metodi* (o operazioni) che l’oggetto può eseguire. Questo modo di vedere l’elaborazione dei dati consente di raggiungere molteplici obiettivi e si basa su alcuni principi di progettazione, di cui parleremo in questo capitolo.

2.1.1 Obiettivi della progettazione orientata agli oggetti

Quando si realizza un prodotto software bisogna avere come obiettivi la *solidità*, la *flessibilità* e la *possibilità di riutilizzo*, come illustrato nella Figura 2.1.

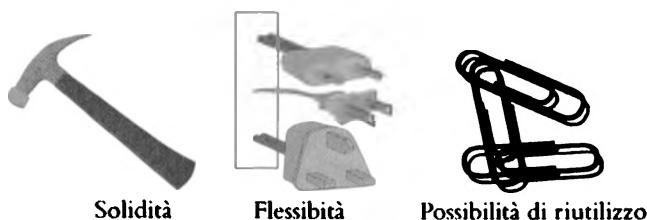


Figura 2.1: Obiettivi della progettazione orientata agli oggetti.

Solidità

Qualsiasi buon programmatore vuole sviluppare software che sia corretto, cioè che produca in uscita i valori previsti per tutti i possibili dati in ingresso di quell'applicazione. In aggiunta, vogliamo che il software sia *solido* o *robusto*, cioè che sia in grado di gestire dati in ingresso anche se inattesi, cioè non esplicitamente previsti: ad esempio, se un programma si aspetta di ricevere un numero intero positivo (magari per rappresentare il prezzo di un articolo) e, invece, viene fornito un numero intero negativo, dovrebbe reagire all'errore senza eccessi, per esempio chiedendo un nuovo valore o terminando la propria esecuzione. Questa proprietà è davvero importante nelle *applicazioni critiche per le funzioni vitali*, dove un errore software può provocare ferite o addirittura portare alla morte: in questi casi un software che non sia robusto può risultare letale, come è accaduto alla fine degli anni Ottanta in alcuni incidenti che hanno coinvolto il Therac-25, un apparato per la radioterapia che, tra il 1985 e il 1987, ha sottoposto a sovradosaggi estremi sei pazienti, alcuni dei quali sono deceduti per le complicazioni derivanti da tale sovradosaggio di radiazioni. In tutti i sei casi si è potuto risalire a errori del software di gestione della macchina.

Flessibilità

Le più moderne applicazioni software, come i *browser* o navigatori per il Web e i motori di ricerca per Internet, sono tipicamente costituite da programmi di grandi dimensioni, che vengono utilizzati per molti anni. Il software, quindi, deve essere in grado di evolversi nel tempo, per rispondere a cambiamenti delle condizioni in cui si trova a operare. Di conseguenza, un altro importante obiettivo da perseguire per la qualità del software è la *flessibilità* (a volte detta anche *possibilità di evoluzione*), un concetto spesso correlato alla *portabilità*, che è la capacità del software di funzionare con modifiche minime su calcolatori dotati di hardware diverso e/o di un diverso sistema operativo. Uno dei vantaggi derivanti dal fatto di scrivere software in Java, in effetti, sta nella portabilità garantita dal linguaggio stesso.

Possibilità di riutilizzo

Tanto si desidera che il software sia flessibile, quanto che sia possibile riutilizzarlo, cioè si vorrebbe che lo stesso codice potesse fungere da componente in diverse applicazioni. Lo sviluppo di software di qualità può essere veramente costoso, un costo che può in qualche modo essere ridotto se il software viene progettato in modo che sia facile riutilizzarlo in altre applicazioni. Questi riutilizzi, però, vanno analizzati con grande cura, perché, ad esempio, una delle principali cause d'errore nel software del Therac-25 fu il riutilizzo inappropriato del software del Therac-20 (che non era orientato agli oggetti e non era stato progettato per la piattaforma hardware poi adottata per il Therac-25).

2.1.2 Principi della progettazione orientata agli oggetti

I principi fondamentali dell'approccio orientato agli oggetti, che ha lo scopo di facilitare il raggiungimento degli obiettivi appena delineati, sono (Figura 2.2):

- Astrazione
- Incapsulamento
- Modularità

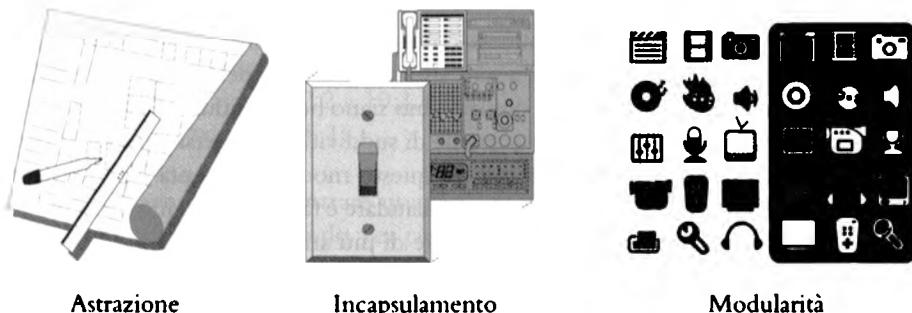


Figura 2.2: Principi della progettazione orientata agli oggetti.

Astrazione

Per **astrazione** si intende il processo di scomposizione di un sistema complesso nelle sue componenti fondamentali e, solitamente, per descrivere le parti di un sistema occorre dar loro un nome e spiegarne le funzionalità. Applicando il paradigma dell'astrazione al progetto di strutture dati si giunge al concetto di *tipo di dato astratto* (*abstract data type*, ADT), che è un modello matematico di una struttura dati che specifica il tipo di dati memorizzati in essa, le operazioni consentite su tali dati all'interno della struttura e il tipo dei parametri richiesti per ciascuna operazione. Un ADT specifica *cosa* fa ciascuna operazione, ma non *come* lo fa. In Java si può descrivere un ADT mediante una *interfaccia*, che è semplicemente un elenco di dichiarazioni di metodi, ciascuno dei quali ha il corpo vuoto (parleremo più diffusamente delle interfacce nel Paragrafo 2.3.1).

Un ADT viene, poi, implementato mediante una struttura dati concreta, che, in Java, si definisce mediante una *classe*. Una classe definisce i dati che sono memorizzati nei suoi esemplari e le operazioni che questi consentono di svolgere su tali dati: diversamente dalle interfacce, le classi descrivono, nel corpo di ciascun metodo, *come* le singole operazioni agiscano sugli oggetti. Si dice che una classe Java *implementa un'interfaccia* ogniqualvolta tra i suoi metodi troviamo tutti i metodi dichiarati nell'interfaccia, dotandoli, in questo modo, di un corpo. Una classe, però, può avere più metodi di quelli richiesti dall'interfaccia.

Incapsulamento

Un altro principio fondamentale su cui si basa la progettazione orientata agli oggetti è l'**incapsulamento**: i vari componenti di un sistema software non dovrebbero rendere noti i dettagli interni della propria implementazione. Uno dei vantaggi principali derivanti dall'incapsulamento è la libertà concessa ai programmatorei nell'implementazione dei dettagli relativi a ciascun componente, senza che si debbano preoccupare del fatto che altri programmatorei possano scrivere codice che dipenda in modo articolato e complesso dalle loro decisioni. L'unico vincolo imposto ai programmatorei di un componente è quello di aderire all'interfaccia pubblica del componente stesso, dal momento che gli altri programmatorei scriveranno il proprio codice basandosi su quella. L'incapsulamento favorisce la solidità e la flessibilità del software, perché consente la modifica dei dettagli realizzativi di parti di un programma senza che questo abbia conseguenze negative su altre parti: in questo modo è più semplice correggere gli errori o aggiungere nuove funzionalità, apportando soltanto modifiche locali a qualche componente.

Modularità

I sistemi software moderni sono tipicamente costituiti da componenti diversi, che devono interagire nel modo corretto perché l'intero sistema funzioni: perché queste interazioni siano ben chiare, è necessario che i vari componenti siano ben organizzati. Per *modularità* si intende un principio organizzativo che prevede di suddividere i diversi componenti di un sistema software in unità funzionali distinte. In questo modo si aumenta notevolmente la solidità del programma, perché è più semplice collaudare e fare il debugging di componenti separati prima di integrarli in un sistema software di più ampie dimensioni.

2.1.3 Schemi di progetto (*design pattern*)

La progettazione orientata agli oggetti agevola la riutilizzabilità, la solidità e la flessibilità del software, ma la scrittura di codice di buon livello richiede ben più della semplice comprensione delle metodologie orientate agli oggetti: necessita dell'effettivo utilizzo delle relative tecniche.

Ricercatori e professionisti dell'informatica hanno sviluppato una gran varietà di concetti e metodologie organizzative per la progettazione orientata agli oggetti di software di qualità, che sia sintetico, corretto e riutilizzabile. Tra queste tecniche, per questo libro è molto rilevante il concetto di *schema progettuale* ricorrente o *design pattern*, che descrive una possibile soluzione per un "tipico" problema di progettazione software. Ognuno di tali schemi costituisce un'infrastruttura generica per una soluzione di un particolare problema, che può, poi, essere applicata in molte situazioni diverse: descrive gli elementi principali della soluzione in modo astratto, consentendone l'adattamento allo specifico problema in esame. Uno schema progettuale è costituito da: un nome, che identifica lo schema; un contesto, che descrive gli scenari in cui si può applicare lo schema; un'infrastruttura (*template*), che descrive come si debba applicare lo schema; un risultato, che descrive e analizza ciò che viene prodotto dallo schema.

In questo libro presenteremo vari schemi progettuali e mostreremo come si possano applicare in modo coerente alla realizzazione di strutture dati e algoritmi. Tali schemi appartengono a due gruppi: schemi che risolvono problemi di progettazione di algoritmi e schemi che risolvono problemi di ingegneria del software. Parleremo, tra gli altri, di questi schemi per algoritmi:

- Ricorsione (Capitolo 5)
- Ammortamento (Paragrafi 7.2.3, 11.4.4 e 14.7.3)
- *Divide-and-conquer* (Paragrafo 12.1.1)
- *Prune-and-search* (Paragrafo 12.5.1)
- Forza bruta (Paragrafo 13.2.1)
- Metodologia *greedy* (Paragrafi 13.4.2, 14.6.2 e 14.7)
- Programmazione dinamica (Paragrafo 13.5)

e di questi schemi per l'ingegneria del software:

- Modello (*template*) di metodo (Paragrafi 2.3.3, 10.5.1 e 11.2.1)
- Composizione (Paragrafi 2.5.2, 2.6 e 9.2.1)
- Adattatore (Paragrafo 6.1.3)

- Posizione (Paragrafi 7.3, 8.1.2 e 14.7.3)
- Iteratore (Paragrafo 7.4)
- Metodo-fabbrica (Paragrafi 8.3.1 e 11.2.1)
- Comparatore (Paragrafi 9.2.2 e 10.3, Capitolo 12)
- Localizzatore (Paragrafo 9.5.1)

Non ha molto senso, però, illustrare qui tutti questi schemi progettuali: li presenteremo nel testo, come indicato, spiegando, per ciascuno di essi (sia per uno schema per algoritmi sia per uno schema per l'ingegneria del software), l'utilizzo generale previsto, insieme con almeno un esempio concreto.

2.2 Ereditarietà

Un modo piuttosto naturale per organizzare i componenti strutturali di un pacchetto software è una *gerarchia*, dove astrazioni simili vengono raggruppate livello per livello, procedendo da descrizioni più specifiche a quelle più generiche a mano a mano che si risale lungo la gerarchia, di cui la Figura 2.3 mostra un esempio. Usando notazioni matematiche, l'insieme delle case è un *sottoinsieme* (*subset*) dell'insieme degli edifici, ma è un *superinsieme* (*superset*) dell'insieme delle case di campagna. Spesso la correlazione tra i livelli viene chiamata *relazione “è un”*, nel senso che una casa è *un* edificio, e una casa di campagna è *una* casa.

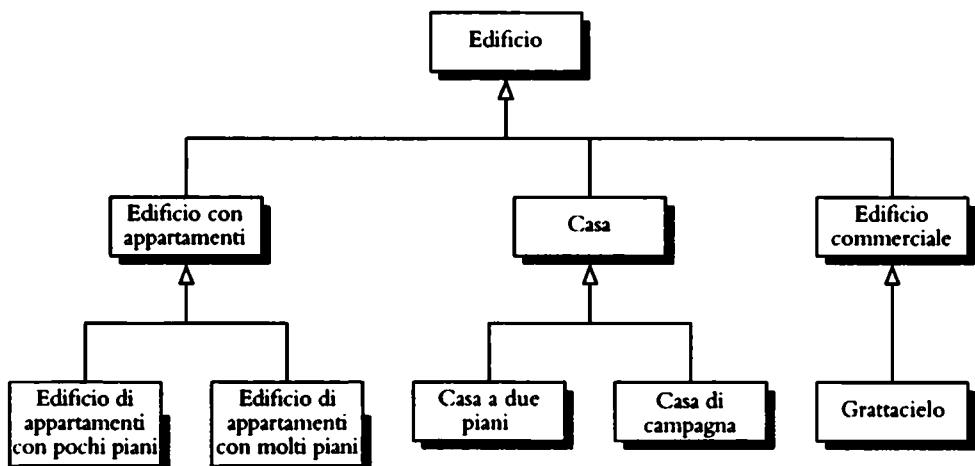


Figura 2.3: Un esempio di gerarchia di tipo “è un” relativa a differenti edifici.

Un progetto gerarchico è di grande utilità nello sviluppo del software, perché una funzionalità comune può essere attribuita a un livello gerarchico più elevato, agevolando così la riutilizzabilità del codice, mentre i comportamenti distintivi di classi simili possono essere visti come estensioni del caso più generale. Nella programmazione orientata agli oggetti, la tecnica che realizza un’organizzazione modulare e gerarchica prende il nome di *ereditarietà* (*inheritance*) e consente di definire una nuova classe che sia basata su un’altra

classe già esistente, presa come punto di partenza. Nella terminologia tradizionale della programmazione orientata agli oggetti, la classe preesistente viene tipicamente descritta come **classe di base**, **classe genitore** o, più spesso, **superclasse**, mentre la classe di nuova definizione è nota come **sottoclasse** o **classe figlia**. Noi diremo che una sottoclassa **estende** la sua superclasse.

Quando si usa l'ereditarietà, la sottoclassa eredita automaticamente, come base di partenza del proprio comportamento, tutti i metodi della superclasse (tranne i costruttori), e può differenziarsi da questa in due modi: può *aggiungere* campi e metodi rispetto alla superclasse, e può *ridefinire* comportamenti ereditati, fornendo una nuova implementazione che *sovrascrive* metodi esistenti.

2.2.1 Estensione della classe CreditCard

Come introduzione all'uso dell'ereditarietà, riprendiamo in esame la classe `CreditCard` vista nel Paragrafo 1.7, progettando una sua sottoclassa che, non avendo trovato un nome migliore che non fosse "da rapina", chiameremo `PredatoryCreditCard`. La nuova classe differirà dall'originale sotto due aspetti: se un tentativo di addebito viene rifiutato perché porterebbe il saldo della carta oltre il suo limite di credito, viene addebitata una commissione di \$5; viene aggiunto un meccanismo per la gestione di un interesse debitore mensile sul saldo non rimborsato, usando un tasso di interesse annuo (APR, *annual percentage rate*) specificato come parametro del costruttore.

La Figura 2.4 mostra lo schema UML che ci servirà come panoramica del progetto della nuova classe `PredatoryCreditCard` come sottoclassa della classe `CreditCard` esistente. La freccia con la punta vuota visibile nello schema indica l'uso dell'ereditarietà e la freccia è orientata dalla sottoclassa alla superclasse.

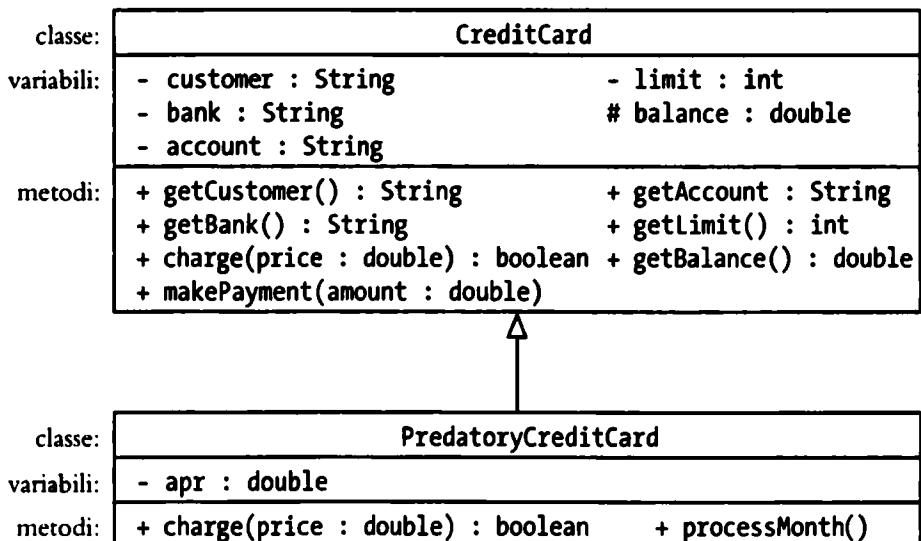


Figura 2.4: Schema UML che mostra la classe `PredatoryCreditCard` come sottoclassa di `CreditCard` (la Figura 1.5 riporta il progetto della classe `CreditCard` originale).

La classe `PredatoryCreditCard` aggiunge alla classe `CreditCard` originale una nuova variabile di esemplare, `apr`, che memorizza il tasso di interesse annuo, e un nuovo metodo, `processMonth`, che si occuperà di addebitare gli interessi mensili. Inoltre, la nuova classe ridefinisce uno dei comportamenti della propria superclasse sovrascrivendo il metodo originale `charge` in modo che abbia una nuova implementazione che addebiti la commissione di \$5 per ogni tentativo di addebito non andato a buon fine.

Per illustrare il funzionamento dell'ereditarietà in Java, il Codice 2.1 presenta un'implementazione completa della nuova classe `PredatoryCreditCard` e, ora, ci soffermeremo su alcuni aspetti di quel codice.

Codice 2.1: Una sottoclasse di `CreditCard` che addebita interessi e commissioni.

```

1  public class PredatoryCreditCard extends CreditCard {
2      // Variabile di esemplare aggiunta
3      private double apr; // tasso di interesse annuo
4
5      // Costruttore di questa classe
6      public PredatoryCreditCard(String cust, String bk, String acnt, int lim,
7                                  double initialBal, double rate) {
8          super(cust, bk, acnt, lim, initialBal); // inizializza la superclasse
9          apr = rate;
10     }
11
12    // Un nuovo metodo per l'addebito degli interessi mensili
13    public void processMonth() {
14        if (balance > 0) { // addebita interessi solo se il saldo è positivo
15            double monthlyFactor = Math.pow(1 + apr, 1.0/12); // calcola il tasso mensile
16            balance *= monthlyFactor; // addebita gli interessi
17        }
18    }
19
20    // Sovrascrive il metodo charge definito nella superclasse
21    public boolean charge(double price) {
22        boolean isSuccess = super.charge(price); // invoca il metodo ereditato
23        if (!isSuccess)
24            balance += 5; // addebita la commissione di $5
25        return isSuccess;
26    }
27 }
```

Iniziamo dalla prima riga della definizione della classe, che, usando la parola chiave di Java `extends` seguita dal nome della superclasse, segnala come la nuova classe sia una sottoclasse della classe `CreditCard` esistente. In Java, ogni classe può estendere una sola altra classe: a causa di questa regola, si dice che Java consente soltanto l'*ereditarietà singola* (e non multipla) tra le classi. Va anche evidenziato che, anche qualora la definizione di una classe non usi in modo esplicito la clausola `extends`, viene automaticamente utilizzata come superclasse la classe `java.lang.Object`, che, per questo motivo, viene detta "superclasse universale" di Java.

Passiamo, ora, alla dichiarazione della nuova variabile di esemplare, `apr`, nella riga 3. Ogni esemplare della classe `PredatoryCreditCard`, quindi, disporrà delle variabili di esemplare ereditate dalla definizione di `CreditCard` (`customer`, `bank`, `account`, `limit` e `balance`), oltre alla nuova variabile `apr`. Come già detto, nella definizione della sottoclasse dobbiamo dichiarare soltanto la nuova variabile di esemplare.

In Java, i costruttori non vengono mai ereditati. Le righe 6–10 del Codice 2.1 definiscono un costruttore della nuova classe. Quando viene creato un esemplare di `PredatoryCreditCard`, tutti i suoi campi devono essere inizializzati in modo appropriato, compresi i campi ereditati. Per questo motivo, la prima operazione eseguita all'interno del corpo del costruttore deve essere l'invocazione del costruttore della superclasse, che ha la responsabilità di inizializzare in modo corretto i campi in essa definiti.

Per invocare uno dei costruttori della superclasse si usa la parola chiave `super` con i parametri appropriati, come si può vedere nella riga 8 della nostra implementazione:

```
super(cust, bk, acnt, lim, initialBal);
```

Questa modalità di utilizzo della parola chiave `super` è molto simile all'uso visto per la parola chiave `this` nel momento in cui, all'interno di una stessa classe, si invoca un diverso costruttore (Paragrafo 1.2.2). Se un costruttore di una sottoclasse non invoca esplicitamente `super` o `this` nella prima riga del suo codice, viene effettuata un'invocazione implicita di `super()`, la versione priva di parametri del costruttore della superclasse. Tornando al costruttore di `PredatoryCreditCard`, dopo aver invocato il costruttore della superclasse con i parametri appropriati, nella riga 9 inizializza il nuovo campo, `apr` (un campo che non è noto alla superclasse).

Il metodo `processMonth` è un comportamento nuovo, per cui non ne esiste una versione ereditata sulla quale basarsi. Nel nostro modello, questo metodo verrà invocato dalla banca, una volta al mese, per addebitare gli interessi al saldo del cliente. Come nota tecnica, osserviamo che questo metodo accede (nella riga 14) al valore del campo `balance` ereditato e, quando opportuno, lo modifica, nella riga 16: questo può avvenire perché il campo `balance` era stato originariamente dichiarato con visibilità `protected` nella classe `CreditCard` (Codice 1.5).

Un aspetto complesso dell'implementazione del metodo `processMonth` riguarda il calcolo del tasso di interesse mensile a partire da quello annuo: non possiamo semplicemente dividere il tasso annuo per dodici, perché si otterrebbe un abuso, con un APR risultante più elevato di quello stabilito. Il calcolo corretto si fa prendendo la radice dodicesima di $1 + \text{apr}$, per poi usare tale valore come fattore moltiplicativo. Se, ad esempio, APR vale 0.0825 (cioè il tasso annuo è 8.25%), calcoliamo $(1.0825)^{1/12} \approx 1.006628$, ottenendo 0.6628% come tasso di interesse mensile. In questo modo, \$100 di debito produrranno correttamente una somma complessiva di \$8.25 come interesse composto durante un intero anno. Si noti l'utilizzo del metodo `Math.pow` della libreria di Java.

Infine, vediamo la nuova implementazione del metodo `charge` definita nella classe `PredatoryCreditCard` (righe 21–27). Questa definizione *sovrascrive* l'omonimo metodo ereditato, ma, come si può notare, l'implementazione del nuovo metodo sfrutta un'invocazione proprio del metodo ereditato, con la sintassi `super.charge(price)` nella riga 22. Il valore restituito da tale invocazione consente di decidere se l'addebito ha avuto successo oppure no: in quest'ultimo caso, verrà addebitata una commissione di \$5 e, in entrambi i casi, quel valore booleano verrà restituito al metodo invocante, in modo che la nuova versione di `charge` mantenga invariata l'interfaccia esterna rispetto alla versione originale.

2.2.2 Polimorfismo e smistamento dinamico

Letteralmente, la parola *polimorfismo* significa "molte forme" e, nel contesto della progettazione orientata agli oggetti, si riferisce al fatto che una variabile riferimento può

assumere forme diverse. Considerate, ad esempio, la dichiarazione di una variabile di tipo `CreditCard`:

```
CreditCard card;
```

Dato che si tratta di una variabile riferimento, l'enunciato dichiara la nuova variabile, senza che questa ancora faccia riferimento a un esemplare di carta di credito. Abbiamo già visto che possiamo assegnare a questa variabile il riferimento a un nuovo esemplare di `CreditCard`, dopo averlo costruito, ma Java consente anche di assegnare a tale variabile il riferimento a un esemplare di `PredatoryCreditCard`, sottoclasse di `CreditCard`, in questo modo:

```
CreditCard card = new PredatoryCreditCard(...); // mancano i parametri
```

Quello appena visto è un esempio del *Principio di Sostituzione di Liskov*, secondo il quale una variabile (o un parametro) che sia dichiarato di un certo tipo può ricevere, come valore, un riferimento a un esemplare di qualunque sottoclasse diretta o indiretta di quel tipo. Detto informalmente, questo è anche un esempio della relazione “è un” rappresentata dall’ereditarietà, perché una carta di credito “da rapina” (esemplare di `PredatoryCreditCard`) è comunque una carta di credito (ma una carta di credito non è necessariamente “da rapina”).

Possiamo, quindi, dire che la variabile `card` è *polimorfica*: può avere una forma o un’altra, in relazione alla particolare classe di cui è esemplare l’oggetto a cui fa riferimento. Dal momento che la variabile `card` è stata dichiarata di tipo `CreditCard`, potrà essere usata soltanto per invocare metodi che facciano parte della definizione di `CreditCard`, per cui potremo scrivere `card.makePayment(50)` o `card.charge(100)`, ma non `card.processMonth()`: l’errore verrebbe segnalato dal compilatore, perché non c’è alcuna garanzia che un esemplare di `CreditCard` disponga del comportamento corrispondente (tale invocazione sarebbe, ovviamente, ammessa se la variabile fosse stata dichiarata di tipo `PredatoryCreditCard`).

È molto importante e interessante capire come Java gestisca un’invocazione come `card.charge(100)` quando la variabile `card` è stata dichiarata di tipo `CreditCard`. Ricordiamo che l’oggetto a cui fa effettivamente riferimento `card` potrebbe essere un esemplare della classe `CreditCard` o della classe `PredatoryCreditCard`, che hanno due diverse implementazioni del metodo `charge`: `CreditCard.charge` e `PredatoryCreditCard.charge`. In situazioni come questa, Java usa una procedura nota come *smistamento dinamico* (*dynamic dispatch*) per decidere, durante l’esecuzione del programma, quale versione del metodo vada invocata: quella più specifica per il tipo effettivo dell’oggetto a cui la variabile fa riferimento (non per il tipo dichiarato per la variabile). Quindi, se l’oggetto è un esemplare di `PredatoryCreditCard`, verrà eseguito il metodo `PredatoryCreditCard.charge`, nonostante la variabile riferimento `card` sia stata dichiarata di tipo `CreditCard`.

Java dispone anche dell’operatore `instanceof`, che verifica, durante l’esecuzione del programma, se un determinato oggetto è un esemplare di un particolare tipo. Ad esempio, la valutazione della condizione booleana (`card instanceof PredatoryCreditCard`) produce il valore `true` se e solo se l’oggetto a cui fa riferimento la variabile `card` è un esemplare della classe `PredatoryCreditCard` o di qualunque ulteriore sottoclasse di tale classe (si veda il Paragrafo 2.5.1 per un ulteriore approfondimento).

2.2.3 Gerarchie di ereditarietà

Nonostante, in Java, una classe non possa ereditare da più superclassi, una superclasse può avere più sottoclassi e, in effetti, non è infrequente che si progettino complesse gerarchie di ereditarietà, per rendere massima la possibilità di riutilizzare il codice.

Come secondo esempio dell'uso dell'ereditarietà, sviluppiamo una gerarchia di classi per la scansione di una progressione numerica. Una progressione numerica è una sequenza di numeri, ciascuno dei quali dipende da uno o più dei numeri che lo precedono nella sequenza stessa. Ad esempio, una *progressione aritmetica* determina il numero successivo aggiungendo una costante prefissata al valore precedente, mentre una *progressione geometrica* determina il numero successivo moltiplicando per una costante prefissata il valore precedente. In generale, una progressione richiede il valore da cui partire (il primo della sequenza) e un modo per identificare il valore successivo sulla base dei precedenti.

La nostra gerarchia deriva da una generica classe di base che chiamiamo `Progression`. Questa classe genera la progressione dei numeri interi: 0, 1, 2, È importante sottolineare che tale classe è stata progettata in modo che sia facile renderla specifica per altri tipi di progressione, dando luogo alla gerarchia visibile nella Figura 2.5.

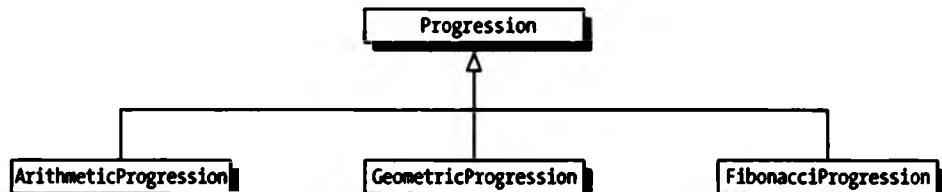


Figura 2.5: La nostra gerarchia di classi che rappresentano progressioni numeriche.

Il Codice 2.2 riporta l'implementazione della classe `Progression`, che ha un unico campo, di nome `current`, e definisce due costruttori: uno che accetta un valore iniziale per la progressione e un altro che usa zero come valore predefinito. La classe, poi, contiene altri tre metodi:

- `nextValue():` Un metodo `public` che restituisce il valore successivo della progressione, procedendo in essa.
- `advance():` Un metodo `protected` che ha il compito di far assumere a `current` il valore successivo nella sequenza.
- `printProgression(n):` Un metodo `public` che fa avanzare `n` volte la progressione, visualizzando `n` valori, uno dopo l'altro così come vengono generati.

Codice 2.2: Classe generica per rappresentare progressioni numeriche.

```

1  /** Genera una semplice progressione: 0, 1, 2, ... */
2  public class Progression {
3
4      // variabile di esemplare
5      protected long current;
6

```

```

7  /** Costruisce una progressione che parte da zero. */
8  public Progression() { this(0); }
9
10 /** Costruisce una progressione con il valore iniziale dato. */
11 public Progression(long start) { current = start; }
12
13 /** Restituisce il valore successivo della progressione. */
14 public long nextValue() {
15     long answer = current;
16     advance(); // questa invocazione fa avanzare la progressione
17     return answer;
18 }
19
20 /** Assegna a current il valore successivo della progressione. */
21 protected void advance() {
22     current++;
23 }
24
25 /** Visualizza i successivi n valori, separati da spazi. */
26 public void printProgression(int n) {
27     System.out.print(nextValue()); // visualizza il primo valore senza spazio
28     for (int j = 1; j < n; j++)
29         System.out.print(" " + nextValue()); // spazio seguito da valore
30     System.out.println(); // va a capo
31 }
32 }
```

La decisione di scorporare il metodo `advance()`, con visibilità `protected`, invocandolo all'interno di `nextValue()`, ha come obiettivo la minimizzazione degli obblighi delle sottoclassi, che devono sovrascrivere solamente `advance` in modo che aggiorni il campo `current`.

Il corpo del metodo `nextValue` memorizza temporaneamente il valore attuale (`current`) della progressione, che verrà restituito al termine del metodo, dopo aver invocato il metodo `advance` (che è `protected`) per aggiornare `current` in preparazione di una futura invocazione.

L'implementazione del metodo `advance` nella classe `Progression` effettua un semplice incremento unitario della variabile `current`: questo è il metodo che verrà sovrascritto dalle sottoclassi per modificare la sequenza dei numeri che vengono generati da una particolare progressione.

Nel seguito di questo paragrafo vedremo tre sottoclassi di `Progression`: `ArithmeticProgression`, `GeometricProgression` e `FibonacciProgression`, che generano, rispettivamente, progressioni aritmetiche, geometriche e di Fibonacci.

Una classe che genera una progressione aritmetica

Il nostro primo esempio di progressione specifica è la progressione aritmetica. Mentre la semplice progressione che abbiamo appena progettato aumenta il proprio valore di un'unità a ogni passo, una progressione aritmetica aggiunge una costante prefissata a un termine della progressione per produrre il successivo. Ad esempio, usando la costante 4 per la progressione aritmetica che parte da 0, si ottiene la sequenza 0, 4, 8, 12,

Il Codice 2.3 presenta la nostra implementazione di una classe `ArithmeticProgression` che usa `Progression` come superclasse. Definiamo tre costruttori, il più generale dei quali (righe 12–15) accetta il valore dell'incremento e il valore iniziale, per cui `ArithmeticProgression(4, 2)` genera la sequenza 2, 6, 10, 14, Il corpo di tale costruttore invoca il costruttore della

superclasse, usando la sintassi `super(start)`, per inizializzare `current` al valore iniziale dato, poi inizializza il campo `increment` definito in questa sottoclass.

Per comodità mettiamo a disposizione altri due costruttori: quello privo di parametri genera la sequenza 0, 1, 2, 3, ..., mentre quello che riceve un solo parametro genera una progressione aritmetica con la costante di incremento data (e valore iniziale uguale a 0).

Infine, la cosa più importante: sovrascriviamo il metodo `protected advance` in modo che il valore di `increment` venga aggiunto a ciascun valore della progressione, uno dopo l'altro.

Codice 2.3: Classe per progressioni aritmetiche, sottoclasse della classe generica per progressioni numeriche vista nel Codice 2.2.

```

1 public class ArithmeticProgression extends Progression {
2
3     protected long increment;
4
5     /** Costruisce la progressione 0, 1, 2, ... */
6     public ArithmeticProgression() { this(1, 0); } // parte da 0 con incremento 1
7
8     /** Costruisce la progressione 0, stepsize, 2*stepsize, ... */
9     public ArithmeticProgression(long stepsize) { this(stepsize, 0); } // parte da 0
10
11    /** Costruisce una progressione aritmetica arbitraria. */
12    public ArithmeticProgression(long stepsize, long start) {
13        super(start);
14        increment = stepsize;
15    }
16
17    /** Aggiunge increment al valore di current. */
18    protected void advance() {
19        current += increment;
20    }
21 }
```

Una classe che genera una progressione geometrica

Il nostro secondo esempio di progressione specifica è la progressione geometrica, nella quale ciascun valore viene generato moltiplicando il valore precedente per una costante prefissata, che prende il nome di `base` della progressione. Il valore iniziale di una progressione geometrica è solitamente 1, e non 0, perché moltiplicando 0 per qualunque fattore si ottiene di nuovo 0. Ad esempio, la progressione geometrica di base 2, partendo dal valore iniziale 1, genera la sequenza 1, 2, 4, 8, 16,

Il Codice 2.4 mostra la nostra implementazione della classe `GeometricProgression`, che è piuttosto simile alla classe `ArithmeticProgression` in merito alle tecniche di programmazione utilizzate. In particolare, definisce un nuovo campo (la base della progressione geometrica), mette a disposizione tre diversi costruttori e sovrscrive il metodo `advance` in modo che il valore della base venga moltiplicato per ciascun valore della progressione, uno dopo l'altro.

Nel caso della progressione geometrica abbiamo deciso che il costruttore privo di parametri usi il valore iniziale 1 e la base 2, in modo da generare la progressione 1, 2, 4, 8, Il costruttore con un solo parametro accetta qualunque valore di base e usa 1 come valore iniziale, in modo che `GeometricProgression(3)` generi la sequenza 1, 3, 9, 27, Infine, il costruttore con due parametri riceve sia la base sia il valore iniziale, per cui `GeometricProgression(3, 2)` costruisce una progressione che genera la sequenza 2, 6, 18, 54,

Codice 2.4: Classe per progressioni geometriche.

```

1  public class GeometricProgression extends Progression {
2
3      protected long base;
4
5      /** Costruisce la progressione 1, 2, 4, 8, 16, ... */
6      public GeometricProgression() { this(2, 1); } // parte da 1 con base 2
7
8      /** Costruisce la progressione 1, b, b^2, b^3, b^4, ... con la base b */
9      public GeometricProgression(long b) { this(b, 1); } // parte da 1
10
11     /** Costruisce una progressione geometrica arbitraria. */
12     public GeometricProgression(long b, long start) {
13         super(start);
14         base = b;
15     }
16
17     /** Moltiplica per base il valore di current. */
18     protected void advance() {
19         current *= base;
20     }
21 }
```

Una classe che genera una progressione di Fibonacci

Come esempio conclusivo, analizziamo l'uso della nostra generica `progressione` per generare una *progressione di Fibonacci*. Ciascun valore di una serie di Fibonacci è la somma dei due valori precedenti della serie. Per iniziare la serie, i primi due valori sono, per convenzione, 0 e 1, dando così luogo alla serie di Fibonacci 0, 1, 1, 2, 3, 5, 8, Più in generale, si può generare questa serie partendo da due valori qualsiasi. Ad esempio, se si parte con i valori 4 e 6, la serie procede come 4, 6, 10, 16, 26, 42,

Il Codice 2.5 presenta un'implementazione della classe `FibonacciProgression`, che è significativamente diversa dalle classi appena viste, per la progressione aritmetica e geometrica, perché non è possibile determinare il successivo valore di una sequenza di Fibonacci sulla base del solo valore attuale: è necessario tenere traccia dei due valori più recenti. Per questo motivo, nella classe `FibonacciProgression` abbiamo introdotto una nuova variabile di esemplare, `prev` (che sta per *previous*, cioè *precedente*), allo scopo di memorizzare il valore che, nella sequenza, precede quello attuale (che è, come al solito, memorizzato nel campo `current` ereditato).

A questo punto, però, sorge un problema, relativo all'inizializzazione del valore `prev` nel costruttore, che riceve come parametri il primo (`first`) e il secondo (`second`) valore con cui dovrà iniziare la sequenza generata. Il primo parametro dovrebbe essere memorizzato in `current`, in modo che venga correttamente restituito dalla prima invocazione di `nextValue()`. Durante tale prima esecuzione di quel metodo, un enunciato di assegnazione farà in modo che il nuovo valore di `current` (che sarà il secondo valore della sequenza) diventi uguale al primo valore sommato al valore di `prev`. Quindi, inizializzando il valore di `prev` a (`second - first`), quel primo enunciato di assegnazione farà in modo che il secondo valore della sequenza sia `first + (second - first)`, cioè `second`, come richiesto.

Codice 2.5: Classe per progressioni di Fibonacci.

```

1  public class FibonacciProgression extends Progression {
2
3      protected long prev;
4
5      /** Costruisce la serie di Fibonacci tradizionale: 0, 1, 1, 2, 3, ... */
6      public FibonacciProgression() { this(0, 1); }
7
8      /** Costruisce una serie di Fibonacci generalizzata, con primo e secondo valore. */
9      public FibonacciProgression(long first, long second) {
10          super(first);
11          prev = second - first;
12      }
13
14      /** Sostituisce (prev,current) con (current,current+prev). */
15      protected void advance() {
16          long temp = prev;
17          prev = current;
18          current += temp;
19      }
20  }

```

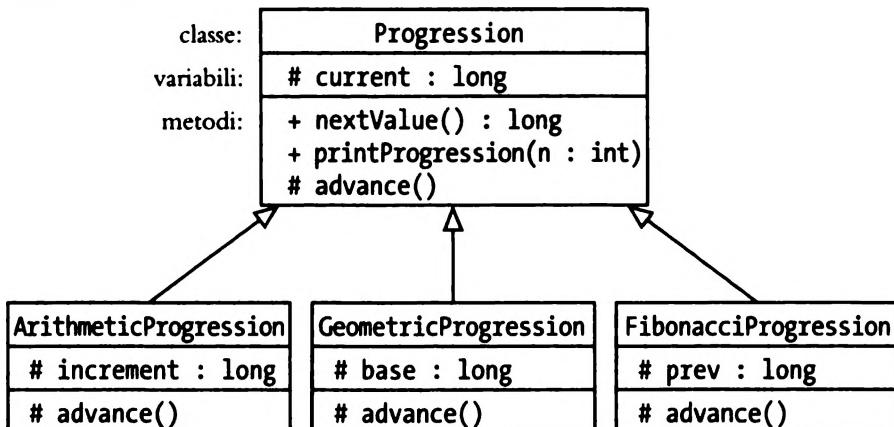


Figura 2.6: Schema di ereditarietà dettagliato con la classe Progression e le sue sottoclassi.

Come riassunto, la Figura 2.6 mostra una versione dettagliata del progetto della gerarchia di ereditarietà, già descritta sommariamente nella Figura 2.5. Si noti come ciascuna classe abbia un campo aggiuntivo, che consente di realizzare correttamente il metodo `advance()` che caratterizza il comportamento di ciascuna progressione.

Collaudo della gerarchia di progressioni

Per completare l'esempio, definiamo nel Codice 2.6 una classe `TestProgression`, che esegue un semplice collaudo di ciascuna delle tre classi. In essa, la variabile `prog` è polimorfica durante l'esecuzione del metodo `main`, perché fa riferimento, di volta in volta, a esemplari di `ArithmeticProgression`, `GeometricProgression` e `FibonacciProgression`. Quando il metodo `main` della classe `TestProgression` viene invocato tramite l'ambiente di esecuzione Java, viene visualizzato quanto riportato nel Codice 2.7.

Codice 2.6: Programma per il collaudo delle classi che rappresentano progressioni.

```

1  /** Programma di collaudo per la gerarchia di progressioni. */
2  public class TestProgression {
3      public static void main(String[] args) {
4          Progression prog;
5          // collaudo ArithmeticProgression
6          System.out.print("Arithmetic progression with default increment: ");
7          prog = new ArithmeticProgression();
8          prog.printProgression(10);
9          System.out.print("Arithmetic progression with increment 5: ");
10         prog = new ArithmeticProgression(5);
11         prog.printProgression(10);
12         System.out.print("Arithmetic progression with start 2: ");
13         prog = new ArithmeticProgression(5, 2);
14         prog.printProgression(10);
15         // collaudo GeometricProgression
16         System.out.print("Geometric progression with default base: ");
17         prog = new GeometricProgression();
18         prog.printProgression(10);
19         System.out.print("Geometric progression with base 3: ");
20         prog = new GeometricProgression(3);
21         prog.printProgression(10);
22         // collaudo FibonacciProgression
23         System.out.print("Fibonacci progression with default start values: ");
24         prog = new FibonacciProgression();
25         prog.printProgression(10);
26         System.out.print("Fibonacci progression with start values 4 and 6: ");
27         prog = new FibonacciProgression(4, 6);
28         prog.printProgression(8);
29     }
30 }
```

Codice 2.7: Informazioni visualizzate dal programma TestProgression del Codice 2.6.

```

Arithmetic progression with default increment: 0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5: 0 5 10 15 20 25 30 35 40 45
Arithmetic progression with start 2: 2 7 12 17 22 27 32 37 42 47
Geometric progression with default base: 1 2 4 8 16 32 64 128 256 512
Geometric progression with base 3: 1 3 9 27 81 243 729 2187 6561 19683
Fibonacci progression with default start values: 0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start values 4 and 6: 4 6 10 16 26 42 68 110
```

L'esempio discusso in questo paragrafo è dichiaratamente semplice, ma presenta alcune delle caratteristiche tipiche di una gerarchia di ereditarietà in Java. Come interessante nota a margine, possiamo notare quanto velocemente crescano i numeri appartenenti alle tre progressioni e quante iterazioni siano necessarie perché i numeri interi di tipo `long` usati per i calcoli siano soggetti a *overflow*. Con l'incremento predefinito al valore 1, una progressione aritmetica va in overflow dopo 2^{63} passi (cioè circa 10 miliardi di miliardi). Una progressione geometrica con base $b = 3$, invece, va in overflow dopo 40 iterazioni, perché $3^{40} > 2^{63}$. Analogamente, il 94-esimo numero di Fibonacci è maggiore di 2^{63} , quindi la progressione di Fibonacci va in overflow dopo 94 iterazioni.

2.3 Interfacce e classi astratte

Perché due oggetti possano interagire, ciascuno di essi deve “conoscere” i diversi “messaggi” che l’altro accetta e comprende, cioè i suoi metodi. Per rendere possibile questa “conoscenza”, il paradigma di progettazione orientata agli oggetti prevede che le classi specifichino la propria *interfaccia per la programmazione di applicazioni* (API, *application programming interface*) o semplicemente *interfaccia*: l’insieme dei comportamenti che i loro esemplari rendono disponibili ad altri oggetti. Nell’approccio alle strutture dati *basato su ADT* (presentato nel Paragrafo 2.1.2) che viene seguito in questo libro, un’interfaccia che definisce un ADT viene specificata mediante la definizione di un tipo di dati e un insieme di metodi con cui poter elaborare dati di quel tipo, dove gli argomenti per ciascun metodo sono di tipi ben specificati. Il rispetto di queste specifiche, poi, viene garantito dal compilatore o dall’ambiente di esecuzione, che richiedono la rigida conformità tra i tipi dei parametri che vengono effettivamente passati ai metodi e i corrispondenti tipi indicati nell’interfaccia. Tutto questo prende il nome di *tipizzazione forte* (*strong typing*). È fuor di dubbio che, per il programmatore, sia un po’ pesante dover definire interfacce, perché poi queste vengano fatte rispettare mediante la tipizzazione forte, ma tutto questo lavoro viene ricompensato dai benefici che ne derivano, perché costringe a seguire il principio dell’incapsulamento e spesso identifica errori di programmazione che, altrimenti, passerebbero inosservati.

2.3.1 Interfacce in Java

Il principale elemento strutturale, in Java, che aiuta a definire una API è l’*interfaccia*, che è un insieme di dichiarazioni di metodi senza un corpo e senza la definizione di dati. In pratica, i metodi di un’interfaccia hanno sempre un corpo vuoto: sono soltanto firme di metodi. Le interfacce non hanno costruttori e non se ne possono creare esemplari.

Quando una classe implementa un’interfaccia, deve implementare tutti i metodi dichiarati nell’interfaccia stessa. In questo modo, le interfacce garantiscono il soddisfacimento del seguente requisito: una classe che implementa un’interfaccia possiede determinati metodi, con una firma ben precisa.

Immaginiamo di voler creare un inventario dei pezzi di antiquariato che possediamo, catalogandoli come oggetti di diverso tipo e con diverse proprietà. Potremmo, ad esempio, voler contrassegnare alcuni di questi oggetti come “vendibili” (*sellable*), nel qual caso dovrebbero implementare l’interfaccia *Sellable* definita nel Codice 2.8.

Poi, possiamo definire una classe concreta, *Photograph* (nel Codice 2.9), che implementa l’interfaccia *Sellable*: ciò significa che potremmo vendere qualunque nostro oggetto di tipo *Photograph*. Questa classe definisce un oggetto che implementa tutti i metodi dell’interfaccia *Sellable*, come richiesto; in aggiunta a ciò, poi, definisce anche un altro metodo, *isColor*, che è specifico degli oggetti di tipo *Photograph*.

Un’altra proprietà degli oggetti presenti nella nostra collezione potrebbe essere il fatto di essere trasportabile. Per tali oggetti, definiamo l’interfaccia che viene presentata nel Codice 2.10.

Codice 2.8: L'interfaccia Sellable.

```

1  /** Interfaccia che descrive oggetti che possono essere venduti. */
2  public interface Sellable {
3
4      /** Restituisce una descrizione dell'oggetto. */
5      public String description();
6
7      /** Restituisce il prezzo in centesimi. */
8      public int listPrice();
9
10     /** Restituisce il prezzo minimo, in centesimi, che verrà accettato. */
11     public int lowestPrice();
12 }

```

Codice 2.9: La classe Photograph che implementa l'interfaccia Sellable.

```

1  /** Classe per fotografie che possono essere vendute. */
2  public class Photograph implements Sellable {
3      private String descript;           // la descrizione di questa fotografia
4      private int price;                // il prezzo richiesto
5      private boolean color;           // true se la fotografia è a colori
6
7      public Photograph(String desc, int p, boolean c) { // costruttore
8          descript = desc;
9          price = p;
10         color = c;
11     }
12
13     public String description() { return descript; }
14     public int listPrice() { return price; }
15     public int lowestPrice() { return price/2; }
16     public boolean isColor() { return color; }
17 }

```

Codice 2.10: L'interfaccia Transportable.

```

1  /** Interfaccia per oggetti che possono essere trasportati. */
2  public interface Transportable {
3      /** Restituisce il peso in grammi. */
4      public int weight();
5      /** Restituisce true se e solo se l'oggetto è pericoloso. */
6      public boolean isHazardous();
7 }

```

A questo punto possiamo definire, nel Codice 2.11, la classe BoxedItem, i cui esemplari rappresentano oggetti d'antichità di vario tipo che possono essere venduti, inscatolati e spediti: tale classe, quindi, implementa tanto i metodi dell'interfaccia Sellable quanto quelli dell'interfaccia Transportable, a cui sono stati aggiunti altri metodi specifici per assegnare alla spedizione un valore da assicurare e per impostare le dimensioni del pacco da spedire.

Codice 2.11: La classe BoxedItem.

```

1  /** Classe per oggetti che possono essere venduti, inscatolati e spediti. */
2  public class BoxedItem implements Sellable, Transportable {
3      private String descript;           // descrizione dell'oggetto

```

```

4  private int price;           // prezzo in centesimi
5  private int weight;          // peso in grammi
6  private boolean haz;         // true se l'oggetto è pericoloso
7  private int height=0;        // altezza della scatola in centimetri
8  private int width=0;         // larghezza della scatola in centimetri
9  private int depth=0;         // profondità della scatola in centimetri
10 /** Costruttore */
11 public BoxedItem(String desc, int p, int w, boolean h) {
12     descript = desc;
13     price = p;
14     weight = w;
15     haz = h;
16 }
17 public String description() { return descript; }
18 public int listPrice() { return price; }
19 public int lowestPrice() { return price/2; }
20 public int weight() { return weight; }
21 public boolean isHazardous() { return haz; }
22 public int insuredValue() { return price*2; }
23 public void setBox(int h, int w, int d) {
24     height = h;
25     width = w;
26     depth = d;
27 }
28 }
```

La classe `BoxedItem` evidenzia un’ulteriore caratteristica delle classi e delle interfacce in Java: una classe può implementare più interfacce (anche se può estendere un’unica altra classe). Questo consente un elevato grado di flessibilità quando si vanno a definire classi che debbono essere conformi a più API.

2.3.2 Ereditarietà multipla per interfacce

La possibilità di estendere più di un tipo di dato è nota come *ereditarietà multipla* e, in Java, l’ereditarietà multipla è consentita soltanto per le interfacce, non per le classi. La motivazione che sta alla base di questa regola sta nel fatto che le interfacce non definiscono campi né corpi di metodi, mentre tipicamente le classi lo fanno. Di conseguenza, se Java consentisse l’ereditarietà multipla per le classi, nascerebbero dei conflitti sintattici nel momento in cui una classe tentasse di estendere due classi che contengono campi aventi lo stesso nome, oppure metodi aventi la stessa firma. Dato che con le interfacce questa confusione non può esserci, Java consente alle interfacce di utilizzare l’ereditarietà multipla, anche perché molte volte questa si rivela utile.

Uno degli utilizzi tipici dell’ereditarietà multipla mediante interfacce consiste nel realizzare un’approximazione di una tecnica di ereditarietà multipla chiamata *mixin* (“miscelezione”). Diversamente da Java, alcuni linguaggi di programmazione orientati agli oggetti, come Smalltalk e C++, consentono l’ereditarietà multipla anche tra classi concrete, non soltanto tramite interfacce. Usando quei linguaggi, è frequente che un programmatore definisca classi (dette “mixin”) che non vengono poi utilizzate per creare esemplari, bensì per fornire funzionalità aggiuntive ad altre classi. Questo tipo di ereditarietà, come già detto, non è consentita in Java, ma i programmatore possono ottenere risultati simili usando le interfacce. In particolare, è possibile utilizzare l’ereditarietà multipla mediante interfacce

come meccanismo per "miscelare" i metodi di due o più interfacce tra loro non correlate, in modo da definire un'interfaccia che combini le loro funzionalità, eventualmente aggiungendo alcuni metodi propri. Tornando al nostro esempio relativo agli oggetti d'antiquariato, potremmo definire un'interfaccia che descriva oggetti assicurabili (in relazione alla loro spedizione per la vendita), in questo modo:

```
public interface Insurable extends Sellable, Transportable {
    /** Restituisce il valore assicurato, in centesimi */
    public int insuredValue();
}
```

Questa interfaccia unisce i metodi dell'interfaccia `Transportable` a quelli dell'interfaccia `Sellable`, e aggiunge all'insieme risultante un ulteriore metodo, `insuredValue`. L'interfaccia `Insurable` così definita ci consentirebbe di definire la classe `BoxedItem` in modo diverso da quanto fatto in precedenza:

```
public class BoxedItem2 extends Insurable {
    // ... codice identico a quello della versione precedente
}
```

Si noti che, in questo caso, il metodo `insuredValue` non è più facoltativo, nonostante lo fosse nella precedente dichiarazione di `BoxedItem`.

Tra le interfacce della libreria Java che approssimano il paradigma "mixin" citiamo `java.lang.Cloneable`, che aggiunge alla classe la possibilità di clonare propri esemplari, `java.lang.Comparable`, che aggiunge alla classe la possibilità di fare confronti tra propri esemplari (imponendo, così, un "ordinamento naturale" all'insieme di tutti i propri esemplari), e `java.util.Observer`, che aggiunge una funzionalità di aggiornamento a una classe che desideri ricevere una notifica ogni volta che determinati oggetti "osservabili" cambiano il proprio stato.

2.3.3 Classi astratte

In Java, una *classe astratta* ha un ruolo in qualche modo intermedio tra quello di una classe tradizionale e quello di un'interfaccia. Come un'interfaccia, una classe astratta può definire firme di metodi senza fornirne l'implementazione (cioè il corpo): saranno *metodi astratti*. Tuttavia, diversamente da un'interfaccia, una classe astratta può definire campi e metodi dotati di implementazione (i cosiddetti *metodi concreti*). Infine, una classe astratta può anche estendere un'altra classe, oltre che essere estesa da sottoclassi.

Come nel caso delle interfacce, non si possono creare esemplari di una classe astratta, che, in un certo senso, è una classe incompleta. Una sottoclasse di una classe astratta deve fornire l'implementazione di tutti i metodi astratti della propria superclasse, oppure rimarrà a sua volta astratta. Per distinguere dalle classi astratte le classi non astratte, chiameremo queste ultime *classi concrete*.

Confrontando i possibili utilizzi delle interfacce e delle classi astratte, è evidente che le classi astratte sono più potenti, dal momento che possono definire anche alcune funzionalità concrete, ma pongono anche più vincoli, perché l'uso delle classi astratte, in Java,

è soggetto alla regola dell'*ereditarietà singola*, per cui una classe può avere al massimo una sola superclasse, sia essa concreta o astratta (Paragrafo 2.3.2).

Nello studio delle strutture dati sfrutteremo molto i vantaggi derivanti dall'utilizzo delle classi astratte, perché agevolano il riutilizzo del codice, uno degli obiettivi della progettazione orientata agli oggetti, come visto nel Paragrafo 2.1.1. Le funzionalità comuni a una famiglia di classi possono essere definite in una classe astratta, che serva da superclasse per più classi concrete. In questo modo, le sottoclassi concrete devono soltanto realizzare quelle funzionalità aggiuntive che differenziano una classe dall'altra.

Come esempio concreto, riprendiamo in esame la gerarchia di progressioni numeriche presentata nel Paragrafo 2.2.3. Pur non avendo, in quel momento, dichiarato la classe `Progression` come astratta, sarebbe stato ragionevole farlo.¹ Infatti, non è previsto che si creino esemplari della classe `Progression`, dal momento che la sequenza che essi genererebbero è semplicemente una progressione aritmetica con incremento unitario. Lo scopo principale dell'esistenza della classe `Progression` è la definizione di funzionalità comuni alle sue tre sottoclassi: la dichiarazione e inizializzazione del campo `current` e l'implementazione concreta dei metodi `nextValue` e `printProgression`.

Nella definizione di una sottoclasse specializzata di `Progression`, il punto qualificante è la sovrascrittura del metodo `advance`, con visibilità `protected`. Sebbene nella classe `Progression` sia stata fornita un'implementazione di tale metodo, che incrementa di un'unità il valore di `current`, nessuna delle tre sottoclassi sfrutta tale implementazione. Nel seguito, illustreremo l'utilizzo delle classi astratte in Java, riprogettando la classe di base `Progression` e trasformandola nella classe astratta `AbstractProgression`. In questo nuovo progetto, il metodo `advance` rimane astratto, in modo che le sottoclassi debbano necessariamente assumersi l'onere di implementarlo.

Funzionamento delle classi astratte in Java

Nel Codice 2.12 presentiamo l'implementazione, in Java, di una nuova classe di base astratta per la nostra gerarchia di progressioni numeriche. L'abbiamo chiamata `AbstractProgression`, invece che semplicemente `Progression`, per tenerla più facilmente distinta dalla versione precedente nella discussione qui riportata. Le definizioni sono quasi identiche, soltanto due sono le differenze importanti che vogliamo sottolineare. La prima è l'uso del modificatore `abstract` nella riga 1, ad integrare la dichiarazione della classe (si veda il Paragrafo 1.2.2 per una discussione sui modificatori di classe).

Come nella nostra classe originaria, anche la nuova classe dichiara il campo `current` e definisce i costruttori che lo inizializzano. Anche se di questa classe, essendo astratta, non potranno essere creati esemplari, i costruttori possono essere invocati dai costruttori delle sottoclassi usando la parola chiave `super` (e in effetti lo facciamo, in tutte le tre sottoclassi concrete).

La nuova classe ha la stessa implementazione concreta della classe originale per i metodi `nextValue` e `printProgression`. Tuttavia, definiamo il metodo `advance` usando esplicitamente il modificatore `abstract`, alla riga 19, evitando di scrivere il corpo del metodo.

Pur non avendo implementato il metodo `advance` nella definizione della classe `AbstractProgression`, lo si può lecitamente invocare dall'interno del corpo del metodo `nextValue`: è un esempio di uno schema di progettazione orientata agli oggetti che prende il nome di *schema con modello di metodo* (*template method pattern*), nel quale una classe di base astratta definisce un comportamento concreto (in questo caso, `nextValue`) che dipende dall'invocazione di altri comportamenti astratti (in questo caso, `advance`). Una volta che una sottoclasse

fornisca l'implementazione dei comportamenti astratti mancati, il comportamento concreto ereditato è ben definito.

Codice 2.12: Una versione astratta della classe di base delle progressioni numeriche, originariamente progettata nel Codice 2.2 (abbiamo omesso i commenti di documentazione per brevità).

```

1  public abstract class AbstractProgression {
2      protected long current;
3      public AbstractProgression() { this(0); }
4      public AbstractProgression(long start) { current = start; }
5
6      public long nextValue() {           // metodo concreto
7          long answer = current;
8          advance();    // fa progredire il valore di current
9          return answer;
10     }
11
12     public void printProgression(int n) {   // metodo concreto
13         System.out.print(nextValue());        // senza spazio iniziale
14         for (int j=1; j < n; j++)
15             System.out.print(" " + nextValue()); // spazio tra due valori
16         System.out.println();                // va a capo
17     }
18
19     protected abstract void advance();      // si noti che manca il corpo
20 }
```

2.4 Eccezioni

Le eccezioni sono eventi inattesi che avvengono durante l'esecuzione di un programma. Un'eccezione può essere prodotta da una risorsa indisponibile, da valori inattesi forniti dall'utente in ingresso o, semplicemente, da un errore logico del programmatore. In Java, le eccezioni sono oggetti che possono essere *lanciati (thrown)* dal codice che si trova di fronte a una situazione inattesa, oppure dalla Java Virtual Machine, se, ad esempio, esaurisce la memoria disponibile per il programma. Un'eccezione può essere *catturata (caught)* da un blocco di codice circostante, che “gestisce” poi il problema in modo appropriato. Se, invece, un'eccezione lanciata non viene catturata, provoca il blocco della macchina virtuale, che termina l'esecuzione del programma e visualizza un opportuno messaggio nella finestra di console. In questo paragrafo parleremo dei tipi di eccezioni più comuni in Java, così come della sintassi per lanciare e catturare eccezioni.

2.4.1 Catturare eccezioni

Se si verifica un'eccezione e questa non viene catturata, l'ambiente Java terminerà l'esecuzione del programma, dopo aver visualizzato un opportuno messaggio e una traccia del contenuto del *runtime stack* (cioè della *pila di esecuzione*). Il *runtime stack* contiene, una dopo l'altra in ordine, le invocazioni di metodi che erano attive nel programma nel momento in cui si è verificata l'eccezione, come in questo esempio:

```
Exception in thread "main" java.lang.NullPointerException
at java.util.ArrayList.toArray(ArrayList.java:358)
at net.datastructures.HashChainMap.bucketGet(HashChainMap.java:35)
at net.datastructures.AbstractHashMap.get(AbstractHashMap.java:62)
at dsaj.design.Demonstration.main(Demonstration.java:12)
```

Prima che l'esecuzione del programma venga interrotta, però, ogni metodo presente nel *runtime stack* ha la possibilità di *catturare (catch)* l'espressione lanciata. Partendo dal metodo in cui è stata lanciata l'eccezione, proseguendo con quello più recente tra quelli presenti nel *runtime stack* e così via fino ad arrivare al metodo `main`, ogni metodo può decidere se catturare l'eccezione o lasciarla arrivare al metodo invocante. Ad esempio, nella pila appena trascritta, il metodo `ArrayList.toArray` è stato il primo ad avere l'opportunità di catturare l'eccezione. Dato che non l'ha fatto, l'eccezione è arrivata, scorrendo la pila, al metodo successivo, `HashChainMap.bucketGet`, che, a sua volta, ha ignorato l'eccezione, facendola proseguire oltre, fino al metodo `AbstractHashMap.get`. L'ultimo ad avere avuto l'occasione di catturare l'eccezione è stato il metodo `Demonstration.main`, ma, dato che non l'ha fatto, il programma è terminato con quella segnalazione d'errore.

La metodologia generale per gestire le eccezioni prevede di utilizzare il costrutto sintattico *try-catch*, nel quale una sezione di codice che potrebbe lanciare un'eccezione viene eseguita in modo controllato: se viene effettivamente lanciata un'eccezione, questa viene *catturata* facendo proseguire il programma con l'esecuzione di un blocco `catch`, che contiene codice (scritto dal programmatore) che analizza l'eccezione e mette in atto contromisure adeguate; se, invece, nel codice posto sotto controllo non si verifica alcuna eccezione, tutti i blocchi `catch` vengono ignorati.

La sintassi tipica per un *enunciato try-catch* in Java è la seguente:

```
try {
    corpoControllato
} catch (tipoDiEccezione1, variabile1) {
    corpoContromisura1
} catch (tipoDiEccezione2, variabile2) {
    corpoContromisura2
} ...
...
```

Ogni `tipoDiEccezionei` è il tipo di una eccezione e ogni `variabilei` è un nome di variabile valido in Java.

L'ambiente Java inizia l'esecuzione di un enunciato *try-catch* come questo eseguendo il blocco di enunciati `corpoControllato`. Se durante questa esecuzione non si verifica alcuna eccezione, il flusso prosegue con il primo enunciato che segue l'ultima riga dell'intero enunciato *try-catch*, ignorando tutti i blocchi `catch`.

Se, al contrario, l'esecuzione del blocco `corpoControllato` genera un'eccezione (ovunque essa venga generata), l'esecuzione di quel blocco termina immediatamente e il flusso d'esecuzione salta al blocco `catch` il cui `tipoDiEccezione` corrisponde in modo più preciso all'eccezione lanciata (se ne esiste almeno uno). La `variabile` di tale blocco di enunciati farà, da questo momento in poi, riferimento all'oggetto lanciato (cioè all'eccezione) e può essere utilizzata all'interno del blocco `catch` che viene eseguito. Terminata l'esecuzione di tale

blocco catch, il flusso procede con il primo enunciato che segue l'ultima riga dell'intero enunciato *try-catch*, ignorando tutti gli altri blocchi *catch*.

Se durante l'esecuzione del blocco *corpoControllato* si verifica un'eccezione che non corrisponde a nessuno dei tipi dichiarati nei blocchi *catch* presenti nell'enunciato *try-catch*, tale eccezione viene rilanciata e prosegue nel contesto circostante.

Quando un'eccezione viene catturata, le *contromisure* possibili sono parecchie. Una di queste è la visualizzazione di un messaggio d'errore, per poi terminare comunque il programma. Esistono anche casi in cui la miglior gestione di un'eccezione consiste nel catturarla e ignorarla silenziosamente, senza prendere alcuna contromisura (cosa che si può realizzare usando un blocco *catch* con il corpo vuoto). Un altro modo lecito di gestire un'eccezione è la creazione e il lancio di una diversa eccezione, che magari specifichi in modo più preciso la condizione eccezionale che si è verificata.

Completiamo rapidamente il discorso segnalando che gli enunciati *try-catch*, in Java, consentono l'utilizzo di alcune tecniche di gestione delle eccezioni più avanzate di quelle utilizzate in questo libro. Ci può essere una clausola *finally* opzionale, con un corpo che verrà eseguito indipendentemente dal fatto che nel *corpoControllato* si verifichi un'eccezione: può essere utile, ad esempio, per chiudere un *file* prima di procedere oltre. La versione SE 7 di Java ha introdotto una nuova sintassi, nota come "try con risorse", che consente di liberare le risorse, come i *file* aperti, in modo anche più avanzato. Sempre a partire dalla versione SE 7, ogni enunciato *catch* può dichiarare di poter gestire più tipi di eccezioni, mentre, in precedenza, questo si poteva ottenere duplicando il corpo del *catch* e inserendolo più volte, per gestire separatamente i vari tipi di eccezione, anche se la contromisura era sempre la stessa.

Codice 2.13: Un esempio di enunciato *try-catch*.

```

1 public static void main(String[] args) {
2     int n = DEFAULT;
3     try {
4         n = Integer.parseInt(args[0]);
5         if (n <= 0) {
6             System.out.println("n must be positive. Using default.");
7             n = DEFAULT;
8         }
9     } catch (ArrayIndexOutOfBoundsException e) {
10        System.out.println("No argument specified for n. Using default.");
11    } catch (NumberFormatException e) {
12        System.out.println("Invalid integer argument. Using default.");
13    }
14 }
```

Come esempio di enunciato *try-catch*, consideriamo la semplice applicazione presentata nel Codice 2.13. Questo metodo *main* cerca di interpretare come numero intero positivo il primo argomento fornito sulla riga di comando (gli argomenti sulla riga di comando sono stati presentati nel Paragrafo 1.2.2).

L'enunciato che potrebbe lanciare un'eccezione, nella riga 4, è *n = Integer.parseInt(args[0])*. Tale enunciato può fallire per due diverse ragioni. Innanzitutto, il tentativo di accesso *args[0]* fallirà nel caso in cui l'utente non abbia fornito alcun argomento sulla riga di comando e, conseguentemente, l'array *args* sarà vuoto, cioè avrà lunghezza zero. In tal

caso verrà lanciata un'eccezione di tipo `ArrayIndexOutOfBoundsException`, catturata alla riga 9. La seconda potenziale eccezione si verifica in seguito all'invocazione del metodo `Integer.parseInt`, che ha successo soltanto se il parametro fornito è una stringa che rappresenti un numero intero valido, come "2013". Dato che, però, un argomento sulla riga di comando può essere una stringa qualsiasi, l'utente potrebbe fornire una sequenza di caratteri che non sia la rappresentazione di un numero intero, nel qual caso il metodo `parseInt` lancerà un'eccezione di tipo `NumberFormatException`, catturata alla riga 11.

Un'ultima condizione che vogliamo impostare è che il numero intero fornito dall'utente sia positivo: per verificare questa proprietà, usiamo un normale enunciato condizionale (righe 5–8). Si noti, però, che tale enunciato condizionale è stato inserito all'interno del corpo principale dell'enunciato *try-catch*: in questo modo sarà eseguito soltanto se, alla riga 4, la conversione avrà successo senza il verificarsi di un'eccezione. Se alla riga 4 si verifica, invece, un'eccezione, il corpo principale del *try* viene abbandonato e il controllo viene assunto direttamente dal gestore di eccezione definito per quell'eccezione.

Infine, se volessimo usare lo stesso messaggio d'errore per entrambe le cause di lancio di eccezione, potremmo usare un'unica clausola *catch*, con la sintassi seguente:

```
} catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
    System.out.println("Using default value for n.");
}
```

2.4.2 Lanciare eccezioni

Le eccezioni nascono quando una sezione di codice Java identifica un problema, di vario tipo, durante la propria esecuzione e *lancia* un oggetto di tipo eccezione, usando la parola chiave `throw` (appunto, *lanciare*) seguita da un esemplare del tipo di eccezione che si deve lanciare. Spesso risulta comodo creare l'esemplare dell'oggetto da lanciare nel momento stesso in cui lo si lancia, per cui l'enunciato `throw` viene solitamente scritto in questo modo:

```
throw new tipoDiEccezione(parametri);
```

dove `tipoDiEccezione` è il tipo dell'eccezione e i `parametri` sono informazioni da trasferire al costruttore di quel tipo. La maggior parte delle eccezioni ha un costruttore che riceve un messaggio d'errore, cioè un parametro di tipo stringa.

Ad esempio, il metodo seguente riceve come parametro un numero intero, che deve essere positivo. Se è negativo, viene lanciata un'eccezione di tipo `IllegalArgumentException`.

```
public void ensurePositive(int n) {
    if (n < 0)
        throw new IllegalArgumentException("That's not positive!");
    // ...
}
```

L'esecuzione di un enunciato `throw` termina immediatamente l'esecuzione del corpo del metodo.

La clausola throws

Quando si dichiara un metodo, è possibile dichiarare esplicitamente, come parte della sua firma, anche l'eventualità che durante la sua invocazione venga lanciata un'eccezione di un determinato tipo, indipendentemente dal fatto che questa venga lanciata direttamente da un enunciato **throw** presente nel corpo del metodo o si propaghi per effetto dell'invocazione di un altro metodo.

La sintassi per dichiarare nella firma di un metodo le eccezioni che possono essere lanciate all'interno del suo corpo prevede l'utilizzo della parola chiave **throws** (da non confondere con l'enunciato **throw**). Ad esempio, il metodo **parseInt** della classe **Integer** ha questa firma:

```
public static int parseInt(String s) throws NumberFormatException
```

La presenza della clausola “**throws NumberFormatException**” informa i programmatore, utenti di quel metodo, che si può verificare una condizione eccezionale: in questo modo possono predisporre il loro codice per gestire l'eccezione che potrebbe essere lanciata. Se un metodo può lanciare eccezioni di tipi diversi, è possibile elencarli nella stessa clausola, separati da virgole. In alternativa, è possibile citare una superclasse di tutte le eccezioni che possono essere lanciate.

L'uso di una clausola **throws** nella firma di un metodo non esime dalla responsabilità di documentare in modo adeguato tutte le possibili eccezioni lanciate da un metodo mediante il marcatore **@throws** all'interno del commento **javadoc** del metodo (come visto nel Paragrafo 1.9.4), descrivendo, per ciascun tipo di eccezione, i motivi che provocano il suo lancio.

Al contrario, l'uso della clausola **throws** nella firma di un metodo è facoltativo per molti tipi di eccezioni. Ad esempio, la documentazione del metodo **nextInt()** della classe **Scanner** evidenzia chiaramente che si possono verificare tre diversi tipi di eccezione:

- Se lo scanner è stato chiuso, una **IllegalStateException**.
- Se lo scanner è attivo, ma non ci sono token disponibili nel flusso d'ingresso, una **NoSuchElementException**.
- Se il prossimo token disponibile non rappresenta un numero intero, una **InputMismatchException**.

Ciò nonostante, nessuna eccezione è citata nella firma del metodo: sono descritte soltanto nella documentazione.

Per poter comprendere appieno lo scopo della dichiarazione **throws** nella firma di un metodo, è utile approfondire la conoscenza del modo in cui, in Java, è organizzata la gerarchia dei tipi di eccezione.

2.4.3 La gerarchia delle eccezioni in Java

Java definisce una ricca gerarchia di ereditarietà per tutti gli oggetti che sono ritenuti “lanciabili”, cioè di tipo **Throwable**. La Figura 2.7 ne riporta una piccola parte. La gerarchia è stata volontariamente suddivisa in due sottoclassi: **Error** e **Exception**. Gli **errori** sono tipicamente lanciati soltanto dalla Java Virtual Machine e corrispondono alle situazioni più gravi, che solitamente sono anche irrecuperabili, come quando la macchina virtuale si trova a dover

eseguire una classe contenuta in un file non valido, oppure quando il sistema operativo ha esaurito la memoria. Al contrario, le *eccezioni* identificano situazioni alle quali il programma in esecuzione potrebbe ragionevolmente far fronte, come può essere l'impossibilità di aprire un file di dati.

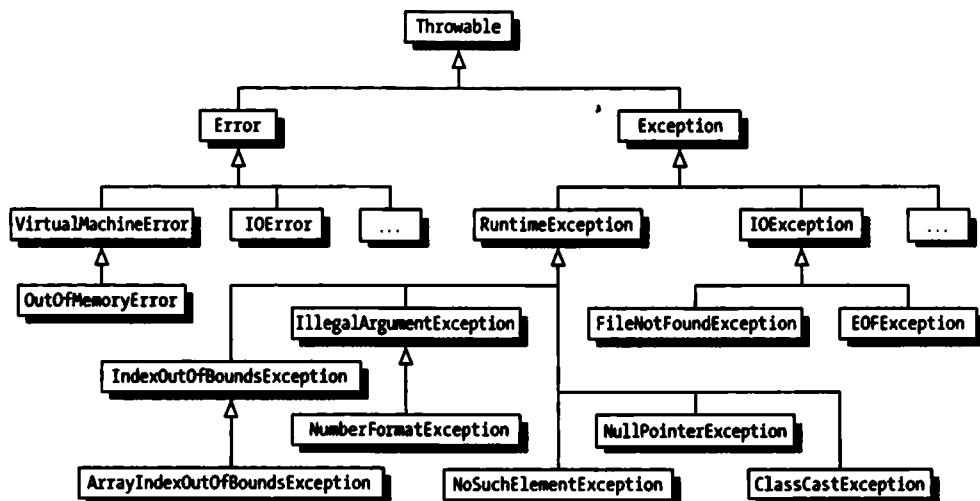


Figura 2.7: Una piccola parte della gerarchia dei tipi `Throwable` in Java.

Eccezioni a controllo obbligatorio e non obbligatorio

Un ulteriore livello di dettaglio, in Java, è dovuto al fatto che la classe `RuntimeException`, sottoclasse di `Exception`, ha un ruolo speciale: tutti i sottotipi di `RuntimeException` sono considerati *eccezioni a controllo non obbligatorio (unchecked exception)*, mentre qualunque altro tipo di eccezione, che non sia sottoclasse diretta o indiretta di `RuntimeException`, è una *eccezione a controllo obbligatorio (checked exception)*.

Lo scopo del progetto di questa gerarchia di eccezioni, così articolata, è che le eccezioni il cui tipo è sottoclasse di `RuntimeException` sono tutte causate da errori logici fatti dal programmatore, come l'uso di un indice non valido in un array o il passaggio di un valore inappropriate come parametro di un metodo. Anche se è sostanzialmente inevitabile che alcuni di tali errori di programmazione si verifichino durante il processo di sviluppo del software, si può presumere che vengano identificati e corretti prima che il software stesso raggiunga una qualità adeguata per entrare in esercizio. Di conseguenza, non è efficiente catturare esplicitamente questo tipo di errori durante l'esecuzione di un programma e, per questo motivo, tali eccezioni sono “a controllo non obbligatorio”.

Al contrario, altre eccezioni si verificano a causa di fenomeni difficilmente controllabili prima che il programma venga eseguito, come l'indisponibilità di un file o l'interruzione di una connessione di rete: sono tipici esempi di eccezioni “a controllo obbligatorio” (e, quindi, non sono sottotipi di `RuntimeException`).

Questa distinzione, tra eccezioni a controllo obbligatorio e non, gioca un ruolo fondamentale nella sintassi del linguaggio. In particolare, tutte le *eccezioni a controllo obbligatorio che si potrebbero propagare a partire da un metodo devono essere esplicitamente dichiarate nella sua firma*.

In conseguenza di ciò, se un metodo ne invoca un altro che dichiara eccezioni a controllo obbligatorio, tale invocazione deve rientrare in un blocco *try-catch* che catturi tali eccezioni, oppure il primo metodo dovrà anch'esso dichiararle nella propria firma, perché c'è il rischio che si propaghino al chiamante.

Definire nuovi tipi di eccezioni

In questo libro ci affideremo completamente ai tipi di eccezioni esistenti come sottoclassi di `RuntimeException` per segnalare le diverse violazioni nell'utilizzo delle strutture dati, ma alcune librerie definiscono nuove classi di eccezioni per poter descrivere condizioni critiche più specifiche. Tali eccezioni dovrebbero essere definite come sottoclassi di `Exception` (se a controllo obbligatorio) o di `RuntimeException` (se a controllo non obbligatorio), oppure di un altro sottotipo di `Exception`, se ne esiste uno più adeguato.

2.5 Cast e tipi generici

In questo paragrafo parleremo di cast tra variabili riferimento e di una tecnica, che prende il nome di “programmazione per tipi generici” o, più brevemente, *generics*, che consente di definire metodi e classi che sono in grado di operare con molti diversi tipi di dati, senza che ci sia bisogno di cast esplicativi.

2.5.1 Cast

Iniziamo la discussione parlando di conversioni di tipo per oggetti.

Conversioni “con ampliamento”

Si verifica una conversione “con ampliamento” (*widening*) quando un tipo *T* viene convertito in un tipo *U* “più ampio”, come in questi casi tipici:

- *T* e *U* sono classi e *U* è una superclasse di *T*.
- *T* e *U* sono interfacce e *U* è una superinterfaccia di *T*.
- *T* è una classe che implementa l’interfaccia *U*.

Le conversioni di questo tipo vengono eseguite automaticamente e permettono di memorizzare il risultato di un’espressione all’interno di una variabile, senza bisogno di un cast esplicito. Quindi, quando la conversione da *T* a *U* è “con ampliamento”, possiamo assegnare direttamente a una variabile *v* di tipo *U* il risultato di un’espressione di tipo *T*. Quando abbiamo parlato di polimorfismo, nel Paragrafo 2.2.2, abbiamo implicitamente utilizzato un cast con ampliamento, assegnando un esemplare della classe `PredatoryCreditCard` a una variabile di tipo `CreditCard`, che era una superclasse di `PredatoryCreditCard`:

```
CreditCard card = new PredatoryCreditCard(...); // mancano i parametri
```

La correttezza di una conversione di questo tipo può essere verificata dal compilatore e la sua validità non necessita di ulteriori verifiche da parte dell’ambiente Java durante l’esecuzione del programma.

Conversioni "con restrizione"

Si verifica una conversione "con restrizione" (*narrowing*) quando un tipo *T* viene convertito in un tipo *S* "più ristretto", come in questi casi tipici:

- *T* e *S* sono classi e *S* è una sottoclasse di *T*.
- *T* e *S* sono interfacce e *S* è una sottointerfaccia di *T*.
- *T* è un'interfaccia implementata dalla classe *S*.

In generale, una conversione di riferimenti che avvenga con restrizione richiede un cast esplicito e la sua correttezza non può essere verificata dal compilatore, per cui deve essere controllata dall'ambiente Java durante l'esecuzione del programma.

L'esempio seguente mostra come usare un cast con restrizione dal tipo `PredatoryCreditCard` al tipo `CreditCard`:

```
CreditCard card = new PredatoryCreditCard(...);      // con ampliamento
PredatoryCreditCard pc = (PredatoryCreditCard) card; // con restrizione
```

Sebbene la variabile `card` faccia effettivamente riferimento a un esemplare di `PredatoryCreditCard`, la variabile è stata dichiarata di tipo `CreditCard`, quindi l'assegnazione `pc = card` è una conversione con restrizione e richiede un cast esplicito, che verrà valutato al momento dell'esecuzione (perché non tutte le carte di credito sono "da rapina").

Eccezioni causate da un cast

In Java, è possibile effettuare un cast da un riferimento di tipo *T* a un riferimento di tipo *S*, a patto che l'oggetto *o* a cui ci si riferisce sia effettivamente di tipo *S*. Se, al contrario, l'oggetto *o* non è anche di tipo *S*, quando si tenta di assegnare il suo riferimento di tipo *T* a una variabile di tipo *S* si verifica un'eccezione di tipo `ClassCastException`. Vediamo questa regola in azione nell'esempio che segue, dove viene usata la classe astratta `Number`, che, in Java, è una superclasse tanto di `Integer` quanto di `Double`:

```
Number n;
Integer i;
n = new Integer(3);
i = (Integer) n; // questo è valido
n = new Double(3.1415);
i = (Integer) n; // questo NON è valido
```

Per evitare questo tipo di problema e per evitare altresì di appesantire il nostro codice con blocchi *try-catch* ogni volta che effettuiamo un cast, Java ci consente di avere la certezza che un cast per un determinato oggetto sarà valido, usando un operatore, `instanceof`, che consente di verificare proprio se una variabile contiene effettivamente un riferimento a un oggetto che sia di un determinato tipo. La sintassi per questo operatore è *riferimentoAOggetto instanceof tipoRiferimento*, dove *riferimentoAOggetto* è un'espressione la cui valutazione genera un riferimento a un oggetto, e *tipoRiferimento* è il nome di una classe, interfaccia o enumerazione esistente. Se *riferimentoAOggetto* fa effettivamente riferimento a un esemplare di una classe compatibile con *tipoRiferimento*, allora l'operatore restituisce `true`, altrimenti

restituisce `false`. Quindi, nell'esempio precedente possiamo evitare che venga lanciata l'eccezione `ClassCastException` scrivendo così:

```
1 Number n;
2 Integer i;
3 n = new Integer(3);
4 if (n instanceof Integer)
5     i = (Integer) n; // questo è valido
6 n = new Double(3.1415);
7 if (n instanceof Integer)
8     i = (Integer) n; // questo NON verrà eseguito!
```

Cast con un'interfaccia

Le interfacce ci consentono di garantire che un oggetto disponga di determinati metodi, ma l'uso di variabili di tipo interfaccia con oggetti concreti richiede a volte operazioni di cast. Supponiamo di aver dichiarato un'interfaccia `Person`, definita nel Codice 2.14. Si noti come il metodo `equals` dell'interfaccia `Person` riceva un parametro di tipo `Person`, per cui possiamo passare a tale metodo un oggetto che sia esemplare di qualunque classe che implementa l'interfaccia `Person`.

Codice 2.14: L'interfaccia `Person`.

```
1 public interface Person {
2     public Boolean equals(Person other); // è la stessa persona?
3     public String getName(); // ispeziona il nome della persona
4     public int getAge(); // ispeziona l'età della persona
5 }
```

Nel Codice 2.15 viene presentata una classe, `Student`, che implementa l'interfaccia `Person`. Dato che il parametro ricevuto da `equals` è di tipo `Person`, l'implementazione del metodo non può assumere che si tratti necessariamente di un oggetto di tipo `Student`, quindi per prima cosa usa l'operatore `instanceof`, alla riga 15, restituendo `false` se l'argomento ricevuto non è uno studente (perché certamente non sarà uguale allo studente in esame). Soltanto dopo aver verificato che il parametro è veramente uno studente, viene eseguito il cast esplicito a `Student`, dopodiché è possibile accedere al suo campo `id`.

Codice 2.15: La classe `Student` che implementa l'interfaccia `Person`.

```
1 public class Student implements Person {
2     String id;
3     String name;
4     int age;
5     public Student(String i, String n, int a) { // un semplice costruttore
6         id = i;
7         name = n;
8         age = a;
9     }
10    protected int studyHours() { return age/2; } // tanto per dire...
11    public String getID() { return id; } // matricola dello studente
12    public String getName() { return name; } // dall'interfaccia Person
13    public int getAge() { return age; } // dall'interfaccia Person
```

```

14     public boolean equals(Person other) {           // dall'interfaccia Person
15         if (!(other instanceof Student)) return false; // non possono essere uguali
16         Student s = (Student) other;                // ora il cast è valido
17         return id.equals(s.id);                     // confronto tra matricole
18     }
19     public String toString() {          // per le visualizzazioni
20         return "Student(ID:" + id + ", Name:" + name + ", Age:" + age + ")";
21     }
22 }
```

2.5.2 Programmazione mediante tipi generici

Il linguaggio Java consente di scrivere classi *generiche* e metodi che possono agire su una varietà di tipi di dati, spesso evitando la necessità di usare cast esplicativi. L'infrastruttura di programmazione per tipi generici (spesso detta semplicemente *generics*) consente di definire una classe in termini di un insieme di *parametri formali di tipo*, che possono essere utilizzati nella definizione della classe per dichiarare tipi delle variabili, dei parametri e dei valori restituiti. Questi parametri formali di tipo verranno specificati in seguito, quando la classe generica sarà utilizzata come tipo di dato all'interno di un programma.

Per giustificare in modo migliore l'utilità dei tipi generici, vediamo un semplice caso di studio. Spesso vogliamo poter trattare una coppia di valori tra loro correlati come se fosse un unico oggetto, ad esempio per fare in modo che un metodo possa restituire una tale coppia. Una soluzione consiste nel definire una nuova classe, i cui esemplari memorizzino entrambi i valori: un primo esempio di uno schema della progettazione orientata agli oggetti che prende il nome di *schema di progettazione mediante composizione (composition design pattern)*. Se sappiamo, ad esempio, che abbiamo bisogno di memorizzare una coppia costituita da una stringa e un numero in virgola mobile, magari per rappresentare il prezzo di un articolo insieme con la sua descrizione, possiamo facilmente progettare una classe apposita per tale scopo. Tuttavia, per un diverso problema, potremmo voler memorizzare una coppia costituita da un oggetto di tipo Book (*libro*) e da un numero intero, che rappresenti la quantità di quei libri presenti in un archivio. La programmazione generica consente di scrivere un'unica classe che possa rappresentare entrambe queste diverse coppie.

L'infrastruttura di programmazione generica non faceva parte del linguaggio Java originario: è stata aggiunta nella versione SE 5. Prima di allora, la programmazione generica si basava pesantemente sulla classe Object di Java, che è il supertipo universale per qualunque oggetto (compresi i tipi involucro, *wrapper*, corrispondenti ai tipi di dati fondamentali). Usando quello stile, ora identificato come “classico”, una generica coppia potrebbe essere implementata come nel Codice 2.16.

Codice 2.16: Definizione di una coppia generica usando lo stile “classico”.

```

1  public class ObjectPair {
2      Object first;
3      Object second;
4      public ObjectPair(Object a, Object b) {      // costruttore
5          first = a;
6          second = b;
7      }
8      public Object getFirst() { return first; }
9      public Object getSecond() { return second; }
10 }
```

Un esemplare di ObjectPair memorizza i due oggetti che vengono passati al suo costruttore e mette a disposizione metodi di accesso per ispezionare i singoli elementi che compongono la coppia. Con questa definizione, l'enunciato seguente dichiara una coppia e ne crea un esemplare:

```
ObjectPair bid = new ObjectPair("ORCL", 32.07);
```

Questa creazione di esemplare è valida perché i parametri forniti al costruttore sono soggetti a conversione per ampliamento. Il primo parametro, "ORCL", è un esemplare di `String`, e quindi anche di `Object`. Il secondo parametro è un valore di tipo `double` che viene automaticamente "avvolto" da un oggetto involucro opportuno, di tipo `Double`, il quale, a sua volta, è valido come `Object` (a dire il vero, questo non è proprio lo stile "classico", perché la tecnica di "auto-boxing" è stata introdotta anch'essa nella versione SE 5 di Java).

Lo svantaggio dell'approccio classico riguarda l'utilizzo dei metodi d'accesso, perché entrambi restituiscono formalmente un riferimento di tipo `Object`. Anche se sappiamo bene che nella nostra applicazione il primo dato contenuto nella coppia è una stringa, non possiamo scrivere questo:

```
String stock = bid.getFirst(); // errore segnalato dal compilatore
```

perché si tratta di una conversione con restrizione dal tipo dichiarato per il valore restituito, `Object`, al tipo della variabile, `String`. Serve, quindi, un cast esplicito, in questo modo:

```
String stock = (String) bid.getFirst(); // conversione con restrizione
```

Usando lo stile classico per realizzare tipi generici, questi cast esplicativi dilagano rapidamente nel codice.

Utilizzo dell'infrastruttura di programmazione generica

In Java, usando l'infrastruttura di programmazione generica (*generics framework*) possiamo realizzare una classe che rappresenti coppie di dati usando i parametri formali di tipo per rappresentare i due tipi di dati che concorrono alla composizione, come si può vedere nel Codice 2.17.

Codice 2.17: Definizione di una coppia usando parametri di tipo generici.

```
1 public class Pair<A,B> {
2     A first;
3     B second;
4     public Pair(A a, B b) {           // costruttore
5         first = a;
6         second = b;
7     }
8     public A getFirst() { return first; }
9     public B getSecond() { return second; }
10 }
```

Nella riga 1 si sono utilizzate le parentesi angolari per racchiudere la sequenza dei parametri formali di tipo. Anche se per rappresentare tali parametri formali si può usare qualunque identificatore valido, convenzionalmente si usano nomi costituiti da un'unica lettera ma-

iuscola (in questo esempio, A e B). Questi parametri di tipo possono, poi, essere utilizzati all'interno del corpo della definizione della classe. Ad esempio, abbiamo dichiarato la variabile di esemplare, first, di tipo A e, analogamente, abbiamo usato A come tipo per il primo parametro del costruttore e per il valore restituito dal metodo `getFirst`.

Al momento dell'utilizzo della classe così definita, dichiarando una variabile di tale tipo, dobbiamo specificare in modo esplicito i *parametri di tipo effettivi (actual type parameters)* che prenderanno il posto dei parametri formali di tipo, generici. Ad esempio, per dichiarare una variabile che faccia riferimento a una coppia contenente il prezzo di un articolo e la sua descrizione, scriveremo così:

```
Pair<String,Double> bid;
```

In questo modo abbiamo affermato di voler usare, per la coppia il cui riferimento verrà memorizzato in `bid`, `String` al posto di A e `Double` al posto di B. I tipi effettivi usati nella programmazione generica non possono essere tipi fondamentali del linguaggio: per questo motivo abbiamo usato la classe involucro `Double` invece del tipo fondamentale `double` (fortunatamente le tecniche di auto-boxing e auto-unboxing giocano a nostro favore).

A questo punto possiamo creare un esemplare della classe generica, in questo modo:

```
bid = new Pair<>("ORCL", 32.07); // sfrutta la deduzione di tipo
```

Dopo l'operatore `new` indichiamo il nome della classe generica, seguito da una coppia di parentesi angolari vuote (coppia vuota che prende il nome di "diamante", *diamond*) e, infine, dai parametri passati al costruttore. Viene così creato un esemplare della classe generica, i cui parametri di tipo effettivi, che vanno a sostituire i parametri di tipo formali, vengono determinati dalla dichiarazione originale della variabile a cui l'esemplare viene assegnato (`bid` in questo esempio). Questa procedura prende il nome di *deduzione o inferenza di tipo (type inference)* ed è stata introdotta nella versione SE 7 di Java.

In alternativa, si può utilizzare lo stile che era in uso prima della versione SE 7, nel quale i parametri effettivi di tipo vengono specificati in modo esplicito tra le parentesi angolari durante la creazione di esemplare, in questo modo:

```
bid = new Pair<String,Double>("ORCL", 32.07); // tipi esplicativi
```

Occorre sottolineare, però, che uno dei due stili deve essere usato: se si omettono completamente le parentesi angolari, come in questo ulteriore esempio:

```
bid = new Pair("ORCL", 32.07); // stile classico
```

si torna allo stile classico, nel quale viene usato automaticamente il tipo `Object` per tutti i parametri di tipo generico, inducendo il compilatore a emettere degli avvertimenti (*warning*) ogni volta che si assegna un valore a una variabile che è di un tipo più specifico.

Anche se la sintassi per la dichiarazione di variabili e la creazione di oggetti usando l'infrastruttura per la programmazione generica è un po' più complessa di quella dello stile classico, il vantaggio che ne deriva riguarda il fatto che non c'è più alcuna necessità di usare i cast esplicativi per le conversioni con restrizione da `Object` a un tipo più specifico. Proseguendo con il nostro esempio, dal momento che `bid` è stata dichiarata usando i parametri effettivi

di tipo `<String,Double>`, il tipo del valore restituito dal metodo `getFirst()` è `String`, mentre il tipo del valore restituito dal metodo `getSecond()` è `Double`. Diversamente da quanto avviene nello stile classico, possiamo scrivere i seguenti enunciati di assegnazione senza usare un cast esplicito (anche se entra in gioco il meccanismo di auto-unboxing relativamente a `Double`):

```
String stock = big.getFirst();
double price = bid.getSecond();
```

Tipi generici e array

In merito all'uso di array di tipi generici, c'è un particolare importante sul quale è bene soffermarsi. Anche se Java consente di definire un array che memorizza un tipo parametrico, dal punto di vista tecnico non è permesso creare nuovi array che coinvolgano tali tipi. Fortunatamente il linguaggio consente di assegnare a una variabile di tipo "array di tipo generico" un esemplare di un array non parametrico, usando un cast opportuno. Anche in questo modo, però, il compilatore emette una segnalazione di potenziale pericolo, perché si tratta di un'azione che non garantisce al 100% il corretto uso dei tipi.

Vedremo questo problema presentarsi in due modi:

- Il codice al di fuori di una classe generica potrebbe voler dichiarare un array che memorizzi riferimenti a esemplari della classe generica, con specifici parametri di tipo effettivi.
- Una classe generica potrebbe voler dichiarare un array che memorizzi riferimenti a oggetti che siano esemplare di uno dei tipi dichiarati come suoi parametri formali.

Come esempio del primo caso, proseguiamo con l'esempio dei prezzi con descrizione e immaginiamo di voler utilizzare un array di oggetti di tipo `Pair<String,Double>`. L'array va dichiarato con un tipo parametrico, ma deve essere inizializzato con un oggetto di tipo *non parametrico*, usando un cast opportuno, come in questo frammento di codice:

```
Pair<String,Double>[] holdings;
holdings = new Pair<String,Double>[25]; // errore in compilazione
holdings = new Pair[25]; // corretto, con warning per il cast
holdings[0] = new Pair<>("ORCL", 32.07); // assegnazione valida
```

Come esempio del secondo caso, immaginiamo di voler progettare una classe generica `Portfolio` che possa memorizzare un certo numero di entità generiche (dette *entry*) in un array. Se la classe usa `<T>` come parametro formale di tipo, può dichiarare un array di tipo `T[]`, ma non ne può creare un esemplare in modo diretto. Una soluzione spesso utilizzata consiste nel creare un array di tipo `Object[]`, per poi assegnarlo alla variabile di tipo `T[]` mediante una conversione con restrizione, come in questo esempio:

```
public class Portfolio<T> {
    T[] data;
    public Portfolio(int capacity) {
        data = new T[capacity];           // errore in compilazione
        data = (T[]) new Object[capacity]; // corretto, con warning
    }
    public T get(int index) { return data[index]; }
    public void set(int index, T element) { data[index] = element; }
}
```

Metodi generici

L'infrastruttura per la programmazione generica consente di definire versioni generiche di singoli metodi (una cosa diversa dalle versioni generiche di intere classi). Per farlo, aggiungiamo ai modificatori del metodo una dichiarazione di parametri formali di tipo.

A titolo di esempio, vediamo una classe non parametrica, `GenericDemo`, dotata di un metodo statico parametrico che è in grado di invertire il contenuto di un array i cui elementi siano oggetti di qualunque tipo.

```
public class GenericDemo {
    public static <T> void reverse(T[] data) {
        int low = 0, high = data.length - 1;
        while (low < high) { // scambia tra loro data[low] e data[high]
            T temp = data[low];
            data[low++] = data[high];
            data[high--] = temp;      // post-decremento di high
        }
    }
}
```

Si noti l'utilizzo del modificatore `<T>` per dichiarare che il metodo è generico, nonché l'uso del tipo `T` all'interno del corpo del metodo, quando viene dichiarata la variabile locale `temp`.

Il metodo può essere invocato usando la sintassi `GenericDemo.reverse(books)`, nel qual caso il meccanismo di deduzione del tipo determinerà il tipo effettivo da usare al posto di quello generico, nell'ipotesi che `books` sia già stata dichiarata come array di qualche tipo di oggetto (questo metodo generico non può essere applicato ad array di tipi fondamentali, perché la tecnica di auto-boxing non si applica a interi array).

Facciamo notare, incidentalmente, che avremmo potuto implementare in modo altrettanto efficiente il metodo `reverse` usando lo stile "classico", elaborando un array di tipo `Object[]`.

Tipi generici vincolati

Se in classi o metodi generici usiamo un nome di tipo parametrico, come `T`, senza specificare nient'altro, questo potrà essere sostituito da un tipo parametrico effettivo qualsiasi. I parametri formali di tipo, però, possono essere *vincolati* usando la parola chiave `extends` seguita dal nome di una classe o di una interfaccia. In tal caso, il parametro formale può essere sostituito soltanto da un parametro effettivo che sia un tipo che soddisfi la condizione dichiarata. Il vantaggio di un tipo vincolato in questo modo è che nel codice generico diventa possibile invocare qualunque metodo la cui esistenza sia garantita dal vincolo espresso.

Ad esempio, potremmo definire una classe generica, `ShoppingCart` (*carrello della spesa*), di cui sia possibile costruire esemplari soltanto fornendo come parametro di tipo effettivo un tipo che implementi l'interfaccia `Sellable` (vista nel Codice 2.8). Tale classe potrebbe essere dichiarata cominciando con questa riga:

```
public class ShoppingCart<T extends Sellable> {
```

Al suo interno potremmo così invocare i metodi `description()`, `listPrice()` e `lowestPrice()` con qualsiasi variabile di tipo `T`.

2.6 Classi annidate

In Java, la definizione di una classe può essere *annidata* (*nested*) all'interno della definizione di un'altra classe. Le classi annidate vengono utilizzate principalmente quando si definisce una classe che è strettamente correlata a un'altra: questa strategia consente di aumentare l'incapsulamento e riduce i conflitti indesiderati tra i nomi. Le classi annidate sono, poi, una tecnica veramente interessante quando si vanno a definire delle strutture dati, perché possono essere usate per rappresentare una piccola parte di una grande struttura o, come classi ausiliarie, possono consentire di navigare all'interno della struttura principale. Infatti, in questo libro useremo le classi annidate per molte implementazioni.

Per illustrare il meccanismo di funzionamento delle classi annidate, prendiamo in esame l'implementazione di una nuova classe, che chiamiamo **Transaction**, che consente di memorizzare le transazioni associate a una carta di credito (un'azione che viene chiamata, in generale, *logging*). La definizione di questa nuova classe può essere annidata all'interno di quella di **CreditCard**, usando lo stile seguente:

```
public class CreditCard {
    private static class Transaction {
        /* omettiamo i dettagli */
    }

    // variabile di esemplare di CreditCard
    Transaction[] history; // transazioni di questa carta
    ...
}
```

La classe più esterna, che contiene la classe annidata, viene detta, appunto, *classe esterna* (*outer class*). Formalmente la classe annidata è un membro della classe esterna e, come tale, il suo nome completo è *NomeClasseEsterna.NomeClasseAnnidata*. Quindi, nell'esempio precedente il nome della classe annidata è **CreditCard.Transaction**, anche se, all'interno della classe **CreditCard**, la possiamo chiamare semplicemente **Transaction**.

L'uso delle classi annidate può aiutare a ridurre il rischio di conflitto tra i nomi, come abbiamo già visto nel caso dei pacchetti (Paragrafo 1.8), perché è perfettamente lecito che ci sia un'altra classe di nome **Transaction** definita come classe annidata di una classe diversa da **CreditCard** (o come classe a sé stante).

Una classe annidata ha un insieme di modificatori indipendente da quello della classe esterna e i suoi modificatori di visibilità determinano se essa sia accessibile all'esterno della definizione della sua classe esterna. Ad esempio, una classe annidata **private** può essere utilizzata soltanto dalla sua classe esterna, ma non da altre classi.

Una classe annidata può anche essere dichiarata **static** oppure **no**, con conseguenze rilevanti. Una classe annidata statica assomiglia molto a una classe tradizionale: i suoi esemplari non hanno alcuna connessione con specifici esemplari della classe esterna.

Una classe annidata non statica viene comunemente chiamata anche *classe interna* (*inner class*) e un esemplare di una classe interna può essere creato soltanto da un metodo non statico della sua classe esterna: l'esemplare della classe interna risulta così essere associato all'esemplare della classe esterna che l'ha creato. Ogni esemplare di una classe interna possiede un riferimento all'esemplare della classe esterna a cui è associato, accessibile dai

metodi della classe interna usando la sintassi *NameClasseEsterna.this* (da non confondere con il semplice *this*, che si riferisce all'esemplare della classe interna). L'esemplare della classe interna ha anche accesso ai membri privati dell'esemplare della classe esterna a cui è associato e, se la classe esterna è generica, può anche utilizzare i suoi parametri formali di tipo.

2.7 Esercizi

Riepilogo e approfondimento

- R-2.1 Fornire tre esempi di applicazioni software di rilevante importanza per la vita umana.
- R-2.2 Fornire un esempio di applicazione software in cui la flessibilità può fare la differenza tra la bancarotta e, invece, vendite significative sul lungo termine.
- R-2.3 Descrivere un componente di un *editor* di testo di una interfaccia grafica (GUI) e i metodi che incapsula.
- R-2.4 Nell'ipotesi di aver modificato la classe *CreditCard* vista nel Codice 1.5 in modo che la sua variabile di esemplare *balance* abbia visibilità **private**, perché la seguente implementazione del metodo *PredatoryCreditCard.charge* è sbagliata?

```
public boolean charge(double price) {
    boolean isSuccess = super.charge(price);
    if (!isSuccess)
        charge(5);           // la commissione da pagare
    return isSuccess;
}
```

- R-2.5 Nell'ipotesi di aver modificato la classe *CreditCard* vista nel Codice 1.5 in modo che la sua variabile di esemplare *balance* abbia visibilità **private**, perché la seguente implementazione del metodo *PredatoryCreditCard.charge* è sbagliata?

```
public boolean charge(double price) {
    boolean isSuccess = super.charge(price);
    if (!isSuccess)
        super.charge(5);      // la commissione da pagare
    return isSuccess;
}
```

- R-2.6 Scrivere un breve frammento di codice Java che usi le classi che rappresentano progressioni numeriche viste nel Paragrafo 2.2.3 per trovare l'ottavo valore di una progressione di Fibonacci che abbia 2 e 2 come primi due valori.
- R-2.7 Se scegliamo un incremento pari a 128, quante invocazioni del metodo *nextValue* della classe *ArithmeticProgression* (vista nel Paragrafo 2.2.3) possiamo effettuare prima che si verifichi una situazione di overflow tra numeri di tipo **long**?
- R-2.8 È possibile che due interfacce si estendano reciprocamente? Motivare la risposta.
- R-2.9 Dal punto di vista dell'efficienza, quali sono i potenziali svantaggi derivanti dall'utilizzo di un albero di ereditarietà estremamente profondo (*deep*), con un insieme di classi, A, B, C, D, ..., tali che B estenda A, C estenda B, D estenda C, e così via?

- R-2.10 Dal punto di vista dell'efficienza, quali sono i potenziali svantaggi derivanti dall'utilizzo di un albero di ereditarietà estremamente poco profondo (*shallow*), con un insieme di classi, A, B, C, D, ..., che estendano tutte la stessa classe Z?
- R-2.11 Cosa viene visualizzato invocando il metodo `main` della classe `Maryland` definita nel frammento di codice seguente, estratto da un determinato pacchetto?

```

public class Maryland extends State {
    Maryland() { /* costruttore vuoto */ }
    public void printMe() { System.out.println("Read it."); }
    public static void main(String[] args) {
        Region east = new State();
        State md = new Maryland();
        Object obj = new Place();
        Place usa = new Region();
        md.printMe();
        east.printMe();
        ((Place) obj).printMe();
        obj = md;
        ((Maryland) obj).printMe();
        obj = usa;
        ((Place) obj).printMe();
        usa = md;
        ((Place) usa).printMe();
    }
}
class State extends Region {
    State() { /* costruttore vuoto */ }
    public void printMe() { System.out.println("Ship it."); }
}
class Region extends Place {
    Region() { /* costruttore vuoto */ }
    public void printMe() { System.out.println("Box it."); }
}
class Place extends Object {
    Place() { /* costruttore vuoto */ }
    public void printMe() { System.out.println("Buy it."); }
}

```

- R-2.12 Disegnare lo schema di ereditarietà per il seguente insieme di classi:

- La classe `Goat` (*capra*) estende `Object` e aggiunge la variabile di esemplare `tail` (*coda*) e i metodi `milk()` (*mungere*) e `jump()` (*saltare*).
- La classe `Pig` (*miaiale*) estende `Object` e aggiunge la variabile di esemplare `nose` (*naso*) e i metodi `eat(food)` (*mangiare cibo*) e `wallow()` (*sguazzare*).
- La classe `Horse` (*cavallo*) estende `Object` e aggiunge le variabili di esemplare `height` (*altezza*) e `color` (*colore*), oltre ai metodi `run()` (*correre*) e `jump()` (*saltare*).
- La classe `Racer` (*corridore*) estende `Horse` e aggiunge il metodo `race()` (*gareggia*).
- La classe `Equestrian` (*equestre*) estende `Horse` e aggiunge le variabili di esemplare `weight` (*peso*) e `isTrained` (*è addestrato*), oltre ai metodi `trot()` (*trottare*) e `isTrained()` (*è addestrato?*). .

- R-2.13 Data la gerarchia di classi dell'Esercizio R-2.12, sia *d* una variabile di tipo `Horse`. Se *d* fa riferimento a un oggetto di tipo `Equestrian`, il suo valore può essere convertito nel tipo `Racer`? Motivare la risposta.

R-2.14 Fornire un esempio di codice Java che accede a un array con un indice che potrebbe essere fuori dai limiti di validità e, in tal caso, la conseguente eccezione viene catturata e viene visualizzato il messaggio seguente:

`"Don't try buffer overflow attacks in Java!"`

R-2.15 Se il parametro fornito al metodo `makePayment` della classe `CreditCard` (vista nel Codice 1.5) fosse un numero negativo, la sua esecuzione avrebbe l'effetto di *aumentare* il saldo del conto. Modificare l'implementazione del metodo in modo che lanci una `IllegalArgumentException` in caso venga fornito come parametro un numero negativo.

Creatività

C-2.16 Pensare a quali potrebbero essere le classi e i metodi principali del progetto in Java di un nuovo lettore di *e-book*. Disegnare uno schema di ereditarietà, senza però scrivere codice. L'architettura del software dovrebbe prevedere quantomeno la possibilità che i clienti acquistino nuovi libri, vedano l'elenco dei libri acquistati e li possano leggere.

C-2.17 Nella maggior parte dei casi, i moderni compilatori per il linguaggio Java dispongono di ottimizzatori che sono in grado di individuare i casi più semplici nei quali risulta logicamente impossibile che determinati enunciati di un programma vengano eseguiti. In tali casi, il compilatore segnala la cosa al programmatore con un *warning*, indicando quale parte del codice sia irraggiungibile dal flusso d'esecuzione e, quindi, inutile. Scrivere un breve metodo Java che contenga codice che senza dubbio non possa essere raggiunto, ma che *non* viene segnalato come tale dal compilatore.

C-2.18 La classe `PredatoryCreditCard` dispone di un metodo, `processMonth()`, che simula il fatto che sia trascorso un mese dalla sua precedente invocazione. Modificare la classe in modo che il proprietario della carta possa richiedere soltanto dieci addebiti (cioè dieci invocazioni di `charge`) durante un mese, dopodiché ciascun ulteriore addebito durante il mese provoca l'addebito di una commissione di \$1.

C-2.19 Modificare la classe `PredatoryCreditCard` in modo che al proprietario della carta sia attribuito un rimborso mensile minimo, come percentuale del saldo. In caso che il rimborso previsto avvenga con più di un mese di ritardo, è previsto l'addebito di una commissione.

C-2.20 Ipotizzare che la classe `CreditCard` (vista nel Codice 1.5) venga modificata in modo che la variabile di esemplare `balance` abbia visibilità `private`, aggiungendo, però, un nuovo metodo `protected`, con firma `setBalance(double newBalance)`, per assegnare un nuovo valore al saldo. Modificare l'implementazione del metodo `PredatoryCreditCard`. `processMonth()` in modo che possa funzionare in questa nuova situazione.

C-2.21 Scrivere un programma costituito da tre classi, *A*, *B* e *C*, tali che *B* estenda *A* e che *C* estenda *B*. Ciascuna classe deve definire una variabile di esemplare di nome *x* (cioè, ciascuna classe ha una propria variabile di nome *x*). Spiegare come un metodo della classe *C* possa assegnare un determinato valore alla variabile *x* della classe *A*, senza modificare il valore delle variabili *x* definite nelle classi *B* e *C*.

C-2.22 Spiegare perché l'algoritmo di smistamento dinamico (*dynamic dispatch*) di Java, che cerca il metodo corretto da eseguire in conseguenza dell'invocazione `obj.foo()`, non entrerà mai in un ciclo infinito.

- C-2.23 Modificare il metodo `advance` della classe `FibonacciProgression` in modo che non usi variabili temporanee.
- C-2.24 Scrivere una classe Java che estenda la classe `Progression` in modo che ciascun valore della progressione generata sia il valore assoluto della differenza tra i due valori precedenti. La classe deve avere un costruttore privo di parametri che generi una sequenza avente 2 e 200 come primi due valori, oltre a un costruttore dotato di due parametri che generi una sequenza che abbia, come primi due valori, i due valori specificati.
- C-2.25 Riprogettare la classe `Progression` in modo che sia astratta e generica, per generare una sequenza di valori del generico tipo `T`, con il supporto per un unico costruttore che accetti un valore iniziale. Modificare in modo coerente tutte le altre classi della gerarchia in modo che continuino a essere classi non generiche, pur ereditando dalla nuova classe `Progression` generica.
- C-2.26 Usare la soluzione dell'Esercizio C-2.25 per creare una nuova classe che rappresenti progressioni in cui ciascun valore sia la radice quadrata del valore precedente, rappresentato sotto forma di `Double`. La classe deve avere un costruttore privo di parametri che generi una sequenza avente 65536 come primo valore, oltre a un costruttore dotato di un parametro che generi una sequenza che abbia come primo valore proprio il parametro.
- C-2.27 Usare la soluzione dell'Esercizio C-2.25 per realizzare una nuova versione della sottoclasse `FibonacciProgression` che usi esemplari della classe `BigInteger`, in modo da evitare completamente il problema dell'overflow.
- C-2.28 Scrivere un insieme di classi Java che simuli un'applicazione Internet in cui un'entità, Alice, crea continuamente insiemi di pacchetti da inviare a un'altra entità, Bob. Un processo di Internet verifica in continuazione se Alice ha pacchetti da inviare e, in caso affermativo, li consegna al computer di Bob, il quale controlla periodicamente se il proprio computer ha ricevuto pacchetti da Alice e, in caso affermativo, li legge e li cancella.
- C-2.29 Scrivere un programma Java che riceve in ingresso un polinomio in notazione algebrica standard e visualizza in uscita la derivata prima di tale polinomio.

Progettazione

- P-2.30 Scrivere un programma Java che riceve in ingresso un documento e visualizza in uscita un grafico a barre relativo alle frequenze delle singole lettere dell'alfabeto all'interno del documento.
- P-2.31 Scrivere un programma Java che simula un ecosistema contenente due tipi di creature viventi, *orsi* e *pesci*. L'ecosistema è costituito da un fiume, il cui modello all'interno del programma è un array relativamente grande. Ogni cella dell'array può contenere un riferimento di tipo `Animal`, che può essere un oggetto di tipo `Bear` (*orso*), un oggetto di tipo `Fish` (*pesce*) o `null`. Ad ogni istante di tempo, sulla base di un processo casuale, ogni animale tenta di spostarsi lungo il fiume, cioè in una delle celle adiacenti alla propria nell'array, oppure rimane dove si trova. Se due animali dello stesso tipo stanno per scontrarsi nella stessa cella, allora rimangono nella cella in cui si trovano, ma viene creato un nuovo esemplare di quel tipo di animale, posizionandolo in una delle celle vuote (cioè contenente `null`) scelta a caso. Se un orso si scontra con un pesce, però, il pesce muore (cioè scompare). Per rappresentare la nascita di animali si usa la normale

- creazione di oggetti, mediante l'operatore `new`. Dopo ogni istante di tempo il programma deve visualizzare il contenuto dell'array.
- P-2.32 Scrivere un simulatore simile a quello dell'Esercizio P-2.31, aggiungendo però a ogni oggetto di tipo `Animal` un campo booleano `gender` (che rappresenta il sesso dell'animale) e un campo `strength` (*forza*) contenente un numero casuale in virgola mobile. Ora, se due animali dello stesso tipo si scontrano in una cella, si crea un nuovo esemplare di quello stesso tipo di animale soltanto se i due animali sono di sesso diverso. Se, invece, si scontrano due animali dello stesso tipo e dello stesso sesso, soltanto il più forte dei due sopravvive.
- P-2.33 Scrivere un programma Java che simula un sistema che abbia le funzioni di un lettore di *e-book*, con metodi che consentano all'utente del sistema di "acquistare" nuovi libri, di vedere la lista dei libri acquistati e di leggerli. Il sistema deve utilizzare veri libri che siano disponibili gratuitamente in Internet (perché i loro diritti d'autore sono scaduti) per popolare l'insieme dei libri disponibili per l'acquisto da parte degli utenti del sistema.
- P-2.34 Definire un'interfaccia `Polygon` che dichiara i metodi `area()` e `perimeter()`. Poi, realizzare le classi `Triangle`, `Quadrilateral`, `Pentagon`, `Hexagon` e `Octagon`, che implementano tutte l'interfaccia `Polygon`. Realizzare anche le classi `IsoscelesTriangle`, `EquilateralTriangle`, `Rectangle` e `Square` (*quadrato*) ciascuna con la corretta relazione di ereditarietà rispetto a una delle precedenti. Infine, scrivere una semplice interfaccia utente che consenta di creare poligoni dei diversi tipi, fornendo le loro dimensioni geometriche, visualizzando poi la loro area e il loro perimetro. Come funzionalità aggiuntiva, consentire all'utente di specificare i poligoni mediante le coordinate cartesiane dei loro vertici e di verificare se due poligoni sono simili.
- P-2.35 Scrivere un programma Java che acquisisce in ingresso un elenco di parole separate da caratteri di spaziatura e visualizza quante volte ciascuna parola compare nella lista. In questo momento, però, non ci si deve preoccupare dell'efficienza, perché è un argomento di cui ci occuperemo più avanti.
- P-2.36 Scrivere un programma Java che "dà il resto". Il programma deve ricevere due numeri in ingresso, che rappresentano la quantità di denaro dovuta da chi deve pagare e la quantità effettivamente consegnata a chi deve ricevere il pagamento. Di conseguenza, il programma poi deve calcolare e visualizzare il numero di ciascun tipo di banconota e moneta da usare per dare il resto, che è ovviamente la differenza tra le due quantità ricevute in ingresso. I valori delle banconote e delle monete disponibili possono essere basati su quelli effettivi di un Paese a scelta. Cercare di progettare il programma in modo che calcoli il minimo numero di banconote e monete possibile.

Note

Per analizzare una panoramica più ampia sugli sviluppi delle scienze informatiche e dell'ingegneria informatica, rimandiamo il lettore a *The Computer Science and Engineering Handbook* [89]. Per maggiori informazioni sull'incidente del Therac-25, si può leggere il lavoro di Leveson e Turner [65].

Il lettore particolarmente interessato allo studio della programmazione orientata agli oggetti può consultare i libri di Booch [16], Budd [19] e Liskov e Guttag [67]. Liskov e Guttag presentano anche un'interessante trattazione dei tipi di dati astratti, come fa anche Demurjian [28] nel suo capitolo all'interno di *The Computer Science and Engineering Handbook* [89]. Gli schemi progettuali (*design pattern*) sono descritti nel libro di Gamma e altri [37].

3

Strutture dati fondamentali

3.1 Array

In questo paragrafo vedremo alcune applicazioni degli array, le strutture dati concrete che abbiamo presentato nel Paragrafo 1.3, nelle quali l'accesso ai dati avviene tramite indici interi.

3.1.1 Memorizzare in un array i punteggi di un gioco

La prima applicazione che analizziamo ha il compito di memorizzare in un array la sequenza dei punteggi migliori ottenuti in un videogioco: si tratta di una situazione simile a quella di molte applicazioni che hanno bisogno di memorizzare una sequenza di oggetti, infatti avremmo potuto analogamente scegliere di memorizzare i dati relativi ai pazienti di un ospedale o i nomi dei giocatori di una squadra di calcio. Concentriamoci, quindi, sulla memorizzazione dei punteggi migliori, un'applicazione abbastanza semplice ma già sufficientemente articolata da consentirci di presentare alcuni concetti importanti sulle strutture dati.

Per iniziare, cerchiamo di capire quali siano le informazioni da inserire in un oggetto che debba rappresentare un dato come “un punteggio ottenuto in un videogioco”. Ovviamente, un dato da inserire è sicuramente un numero intero che rappresenti il punteggio stesso, che chiamiamo `score`. Un altro dato utile è il nome della persona che ha ottenuto il punteggio: identifichiamola con `name`. Potremmo proseguire aggiungendo campi che memorizzino la data di conseguimento del punteggio o informazioni sulle statistiche di gioco che hanno portato a quel punteggio, ma evitiamo di farlo per lavorare su un esempio ragionevolmente semplice. Il Codice 3.1 descrive una classe Java, `GameEntry`, che rappresenta un punteggio in un gioco.

Codice 3.1: Una semplice classe GameEntry: definisce metodi per restituire il nome del giocatore e il punteggio ottenuto, oltre a un metodo che restituisce una stringa che descrive interamente l'oggetto.

```

1  public class GameEntry {
2      private String name;    // nome del giocatore
3      private int score;     // punteggio
4      /** Costruisce una registrazione di punteggio usando i valori forniti. */
5      public GameEntry(String n, int s) {
6          name = n;
7          score = s;
8      }
9      /** Restituisce il nome del giocatore. */
10     public String getName() { return name; }
11     /** Restituisce il punteggio memorizzato. */
12     public int getScore() { return score; }
13     /** Restituisce una stringa che rappresenta questo oggetto. */
14     public String toString() {
15         return "(" + name + ", " + score + ")";
16     }
17 }
```

Una classe per gestire i punteggi migliori

Per gestire un elenco dei punteggi migliori, sviluppiamo la classe Scoreboard (*tabellone segnapunti*). Un segnapunti è caratterizzato da un limite massimo di punteggi migliori che può memorizzare, raggiunto il quale memorizza un nuovo punteggio soltanto se è strettamente migliore del minimo punteggio presente. La lunghezza più adeguata per un tabellone segnapunti dipende dal gioco, ad esempio potrebbe essere 10, 50 o 500. Dato che questo limite è così variabile in base all'utilizzo, permettiamo che venga specificato come parametro per il costruttore di Scoreboard.

Internamente alla classe, memorizzeremo gli oggetti di tipo GameEntry che rappresentano i punteggi migliori in un array di nome board. L'array viene creato in modo che abbia una lunghezza uguale alla massima capacità del segnapunti, ma tutte le sue celle vengono inizializzate a null. Mentre i punteggi vengono aggiunti al segnapunti, li manteniamo ordinati dal punteggio massimo a quello minimo, con il massimo nella cella di indice 0 dell'array. La Figura 3.1 mostra una situazione tipica in cui può trovarsi questa struttura (di memorizzazione dei) dati, mentre il Codice 3.2 riporta il codice Java che la costruisce.

Codice 3.2: La parte iniziale della classe Scoreboard che gestisce un insieme di punteggi sotto forma di oggetti di tipo GameEntry (la classe sarà completata nel Codice 3.3 e 3.4).

```

1  /** Classe che memorizza punteggi in un array in ordine non decrescente. */
2  public class Scoreboard {
3      private int numEntries = 0; // numero di punteggi memorizzati
4      private GameEntry[] board; // array dei dati memorizzati (nome e punteggio)
5      /** Costruisce un segnapunti vuoto con la capacità massima assegnata. */
6      public Scoreboard(int capacity) {
7          board = new GameEntry[capacity];
8      }
9      // qui ci saranno altri metodi
10 }
```

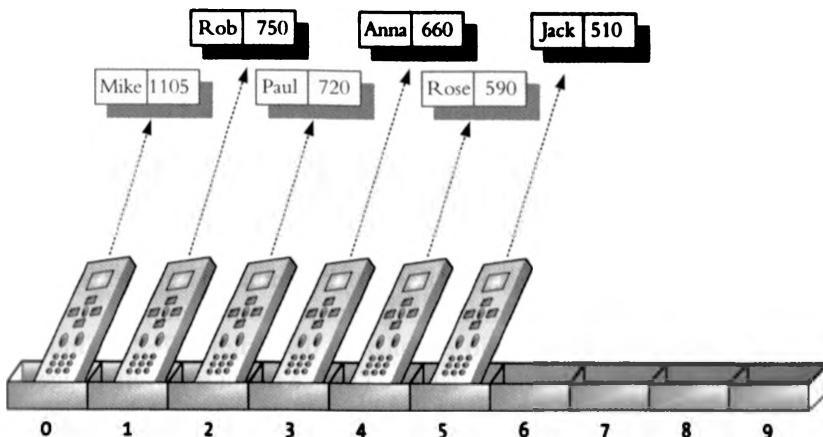


Figura 3.1: Una raffigurazione simbolica dello stato di un array di lunghezza 10 che memorizza riferimenti a 6 oggetti di tipo GameEntry nelle proprie celle aventi indice che va da 0 a 5; gli altri riferimenti sono null.

Aggiungere un punteggio

Una delle azioni di aggiornamento che più spesso vorremo compiere su un oggetto di tipo Scoreboard è l'inserimento di un nuovo punteggio, ricordando, però, che non sempre un nuovo punteggio conseguito da un giocatore avrà i requisiti per diventare un nuovo punteggio nel tabellone segnapunti dei punteggi migliori. Se il segnapunti non è ancora pieno, allora qualunque nuovo punteggio viene memorizzato, ma, una volta che il segnapunti è pieno, un nuovo punteggio viene memorizzato soltanto se è strettamente maggiore di uno di quelli già presenti, in particolare, ovviamente, se è maggiore dell'ultimo punteggio presente nel segnapunti, in quanto questo è proprio quello di valore minimo.

Il Codice 3.3 presenta l'implementazione del metodo di aggiornamento della classe Scoreboard: si occupa dell'inserimento di un nuovo punteggio.

Codice 3.3: Il codice Java per inserire un nuovo oggetto di tipo GameEntry in uno Scoreboard.

```

9  /**
10 * Aggiunge un nuovo punteggio al segnapunti (se è abbastanza alto) */
11 public void add(GameEntry e) {
12     int newScore = e.getScore();
13     // il nuovo punteggio, e, ha diritto di entrare nel segnapunti?
14     if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
15         if (numEntries < board.length)// nessun punteggio deve uscire dal segnapunti
16             numEntries++;           // quindi il numero dei punteggi presenti aumenta
17         // sposta in avanti i punteggi più bassi per far posto al nuovo
18         int j = numEntries - 1;
19         while (j > 0 && board[j-1].getScore() < newScore) {
20             board[j] = board[j-1]; // sposta un dato da j-1 a j
21             j--;                  // poi decrementa j
22         }
23         board[j] = e;           // finito, quindi inserisce il nuovo punteggio
24     }
}

```

Quando viene preso in esame un nuovo punteggio, il primo obiettivo è quello di determinare se entrerà nel segnapunti. Questo avviene (riga 13) se il segnapunti non è pieno oppure se il nuovo punteggio è strettamente maggiore del minimo punteggio presente nel segnapunti.

Dopo aver determinato che il nuovo punteggio deve entrare nel segnapunti, rimangono due cose da fare: aggiornare il numero di punteggi memorizzati e inserire il nuovo dato nella posizione corretta, spostando i punteggi inferiori se necessario.

Il primo di questi compiti viene svolto facilmente dal codice delle righe 14 e 15, perché il numero totale di punteggi presenti deve essere incrementato (di un'unità) solo se il segnapunti non è già pieno (quando è pieno, l'eventuale inserimento di un nuovo punteggio viene compensato dalla rimozione del punteggio minimo).

L'inserimento del nuovo punteggio è realizzato dalle righe 17–22. L'indice j viene inizialmente posto uguale a `numEntries - 1`, che è l'indice in cui si troverà l'oggetto di tipo `GameEntry` contenente il punteggio minimo al termine dell'operazione. Ad ogni iterazione del ciclo `while`, j è l'indice corretto per il nuovo punteggio da inserire oppure il punteggio di indice $j - 1$ è inferiore a tale punteggio: il ciclo verifica tale condizione composta, spostando gli elementi di un posto “verso destra” (cioè verso la posizione di indice immediatamente superiore a quella che occupano) e decrementando j , fino a quando esiste un altro oggetto in corrispondenza dell'indice $j - 1$ con un punteggio inferiore a quello da inserire.

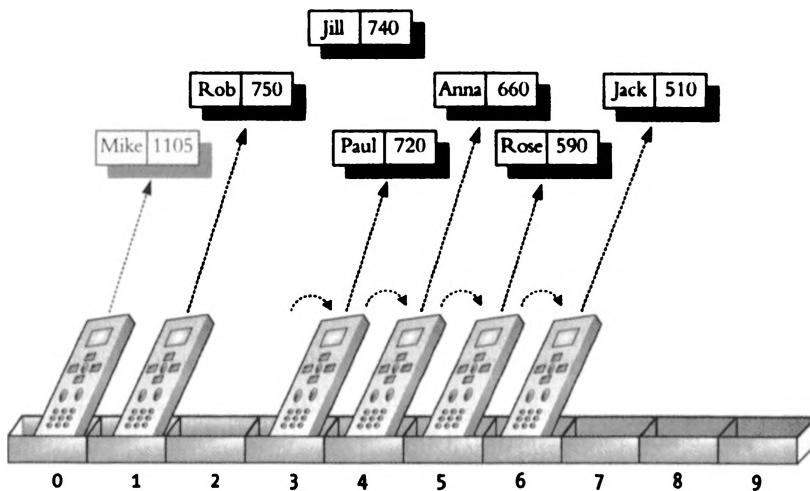


Figura 3.2: Azioni preliminari compiute sull'array `board` per far posto all'oggetto di tipo `GameEntry` contenente il punteggio di Jill: tutti i riferimenti a oggetti contenenti punteggi inferiori a quello di Jill sono stati spostati di una posizione verso destra, cioè verso indici maggiori.

La Figura 3.2 mostra un esempio della procedura di inserimento appena descritta, subito dopo lo spostamento degli elementi che devono essere spostati, ma immediatamente prima dell'inserimento del nuovo punteggio nell'array. Quando il ciclo termina, il valore di j sarà quello corretto per indicare la cella in cui posizionare il nuovo punteggio. La Figura 3.3 mostra il risultato finale dell'operazione, dopo l'assegnazione `board[j] = e` eseguita dalla riga 22 del codice.

Nell'Esercizio C-3.19 vedremo come sia possibile semplificare l'inserimento di nuovi punteggi nel caso in cui non sia necessario preservare il loro ordinamento relativo.

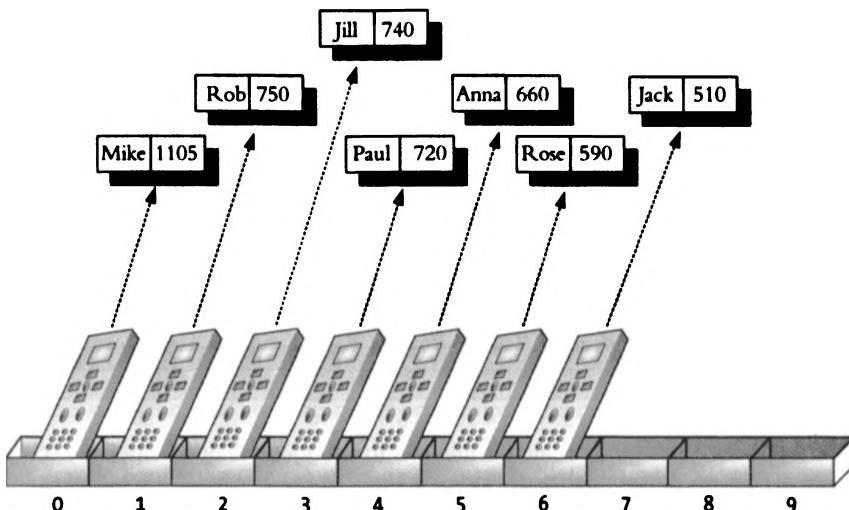


Figura 3.3: Inserimento nell'array board del riferimento all'oggetto di tipo GameEntry contenente il punteggio di Jill; ora può essere inserito nella cella di indice 2, quella corretta, perché tutti i riferimenti a oggetti contenenti punteggi inferiori a quello di Jill sono stati spostati di una posizione verso destra.

Eliminare un punteggio

Supponiamo che alcuni giocatori particolarmente bravi abbiano giocato al nostro videogioco e che i loro punteggi siano stati inseriti nel nostro tabellone segnapunti, salvo poi scoprire che avevano usato dei trucchi non consentiti. In tal caso, sarebbe opportuno disporre di un metodo che ci consenta di eliminare un punteggio dall'elenco dei migliori: vediamo, quindi, come si possa eliminare il riferimento a un determinato oggetto di tipo GameEntry da uno Scoreboard.

Abbiamo deciso di aggiungere alla classe Scoreboard il metodo `remove(i)`, dove i indica l'indice, all'interno dell'array `board`, del punteggio che si vuole eliminare. Quando un punteggio viene rimosso, tutti i punteggi inferiori devono scorrere di una posizione verso gli indici più bassi, in modo da chiudere il buco lasciato libero. Se l'indice i non appartiene all'intervallo degli indici occupati nel segnapunti, il metodo lancia una `IndexOutOfBoundsException`.

La nostra implementazione del metodo `remove` richiede un ciclo per spostare i punteggi, in modo molto simile all'algoritmo usato per l'inserimento, ma in direzione opposta. Per eliminare il riferimento che si trova in posizione i , partiamo dall'indice i e spostiamo di una posizione verso sinistra (cioè verso indici minori) tutti i riferimenti di indice maggiore di i , come si può vedere, ad esempio, nella Figura 3.4.

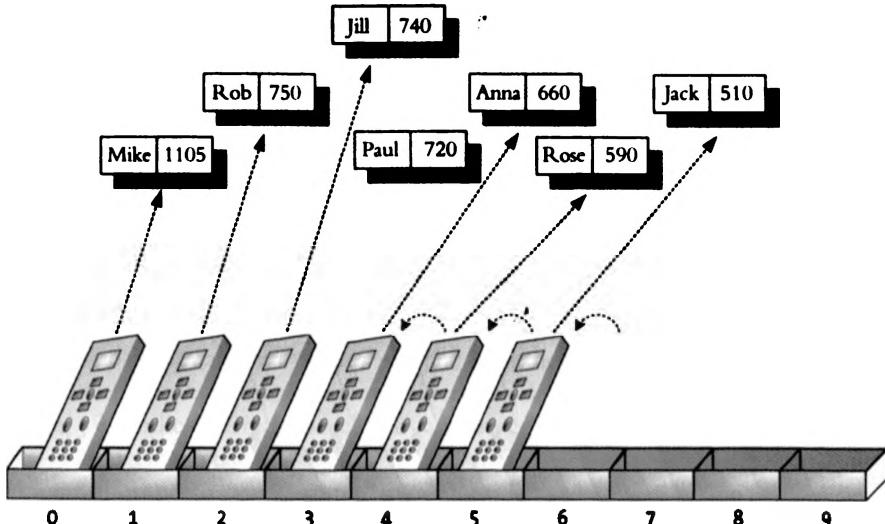


Figura 3.4: Rimozione dall'array `board` del riferimento all'oggetto di tipo `GameEntry` contenente il punteggio di Paul, in corrispondenza dell'indice 3.

Il Codice 3.4 contiene la nostra implementazione del metodo `remove` della classe `Scoreboard`. I dettagli relativi all'operazione di rimozione meritano alcuni chiarimenti, il primo dei quali riguarda il fatto che il metodo deve restituire il riferimento al punteggio rimosso (chiamiamolo `e`), che si trova inizialmente in corrispondenza dell'indice `i` nell'array: per farlo occorre memorizzare `e` in una variabile temporanea, che useremo per restituire il riferimento dopo che sarà stato eliminato dall'array.

Codice 3.4: Il codice Java per eseguire l'operazione `Scoreboard.remove`.

```

25  /** Elimina e restituisce il punteggio che si trova all'indice i. */
26  public GameEntry remove(int i) throws IndexOutOfBoundsException {
27      if (i < 0 || i >= numEntries)
28          throw new IndexOutOfBoundsException("Invalid index: " + i);
29      GameEntry temp = board[i];           // memorizza l'oggetto da rimuovere
30      for (int j = i; j < numEntries - 1; j++) // conta da i in avanti
31          board[j] = board[j+1];           // sposta di un posto verso sinistra
32      board[numEntries - 1] = null;       // cancella l'ultimo punteggio
33      numEntries--;                    // restituisce l'oggetto rimosso
34      return temp;
35  }

```

Il secondo punto che merita un commento riguarda lo spostamento dei riferimenti che si trovano in celle di indice maggiore di `i`, che avviene verso celle di indice inferiore: nel farlo, non procediamo fino alla fine dell'array. Innanzitutto, impostiamo il ciclo sulla base del numero effettivo di punteggi presenti nel segnapunti e non sulla capacità dell'array, perché non c'è motivo per spostare la sequenza di riferimenti `null` che può trovarsi nella porzione terminale dell'array (se il segnapunti non è pieno). Abbiamo anche definito con attenzione la condizione di terminazione del ciclo, `j < numEntries - 1`, in modo che l'ultima iterazione del ciclo effettui l'assegnazione `board[numEntries - 2] = board[numEntries - 1]`.

~~numEntries - 1]. Non c'è nessun punteggio da portare nella cella board[numEntries - 1], per cui subito dopo il ciclo riportiamo tale cella al valore null. Il metodo si conclude restituendo il riferimento al punteggio rimosso (al quale non esiste più alcun riferimento all'interno dell'array board).~~

Conclusioni

I metodi che consentono di aggiungere e rimuovere oggetti in un array di punteggi sono semplici, ma costituiscono la base delle tecniche che vengono ripetutamente utilizzate per costruire strutture dati più complesse. Queste ultime potranno anche avere, ovviamente, un uso più generale dell'array utilizzato qui e spesso disporranno di molte più operazioni da poter eseguire, oltre a add e remove, ma lo studio della struttura dati concreta che chiamiamo array, così come lo stiamo facendo, è un ottimo punto di partenza per comprendere tali diverse strutture, che andranno comunque realizzate a partire da basi concrete.

Più avanti nel libro studieremo la classe `ArrayList` della libreria standard di Java che può essere utilizzata per creare raccolte di dati (*collection*), decisamente più generale dell'array che stiamo studiando. La classe `ArrayList` ha metodi che operano su un array sottostante, ma evita che si verifichi la condizione di errore conseguente al tentativo di aggiungere un oggetto a un array pieno, copiando automaticamente tutto il proprio contenuto in un array più grande ogni volta che ciò si rende necessario. Nel Paragrafo 7.2 analizzeremo in dettaglio la classe `ArrayList`.

3.1.2 Ordinare un array

Nel sottoparagrafo precedente abbiamo analizzato un'applicazione che aggiungeva un oggetto a un array in una determinata posizione, facendo scorrere di un posto altri elementi in modo da preservarne l'ordinamento. In questo paragrafo usiamo una tecnica simile per risolvere il problema dell'*ordinamento* (*sorting*): partendo da un array di elementi non in ordine, vogliamo sistemarli in modo che siano in ordine di valore non decrescente al crescere dell'indice.

L'algoritmo di ordinamento per inserimento

In questo libro vedremo diversi algoritmi di ordinamento, la maggior parte dei quali sono descritti nel Capitolo 12. In questo paragrafo ne vediamo un assaggio: un semplice algoritmo di ordinamento che prende il nome di *ordinamento per inserimento* (*insertion sort*). L'algoritmo procede esaminando un elemento alla volta, posizionandolo correttamente rispetto agli elementi che lo precedono, cioè che si trovano in corrispondenza di indici minori. Partiamo con il primo elemento dell'array, che, banalmente, si trova nel posto giusto rispetto agli elementi che lo precedono, dato che non ce ne sono. Considerando, poi, l'elemento successivo, se è minore del primo elemento, li scambiamo tra loro. Passando all'elemento successivo, il terzo, lo scambiamo con l'elemento alla sua sinistra, e così via facendolo procedere verso sinistra, fino a quando non si trova nella posizione corretta relativamente ai primi due elementi che lo precedevano nell'array. Continuiamo in questo modo con il quarto elemento, il quinto, e così via, fino ad aver ordinato l'intero array. Possiamo esprimere l'algoritmo di ordinamento per inserimento mediante pseudocodice, come si può vedere nel Codice 3.5.

Codice 3.5: Descrizione ad alto livello dell'algoritmo di ordinamento per inserimento.

Algoritmo InsertionSort(A):

Input: Un array A di n elementi confrontabili

Output: L'array A con gli elementi disposti in ordine non decrescente

for k da 1 a $n - 1$ do

Inserisci $A[k]$ nella posizione corretta nella porzione di array $A[0], A[1], \dots, A[k]$.

Questa è una semplice descrizione ad alto livello dell'ordinamento per inserimento. Se torniamo a esaminare il Codice 3.3 del Paragrafo 3.1.1, vediamo che l'inserimento di un nuovo punteggio in un segnapunti è un problema quasi identico all'inserimento di uno degli elementi presi in esame da *insertion sort*, tranne per il fatto che i punteggi venivano ordinati in senso inverso, dal più basso al più alto. Nel Codice 3.6 forniamo un'implementazione in Java dell'ordinamento per inserimento, usando un ciclo esterno che esamina gli elementi uno dopo l'altro e un ciclo interno che sposta l'elemento in esame nella sua posizione corretta relativamente al sottoarray (ordinato) degli elementi che stanno inizialmente alla sua sinistra. La Figura 3.5 mostra un esempio di esecuzione dell'algoritmo di ordinamento per inserimento.

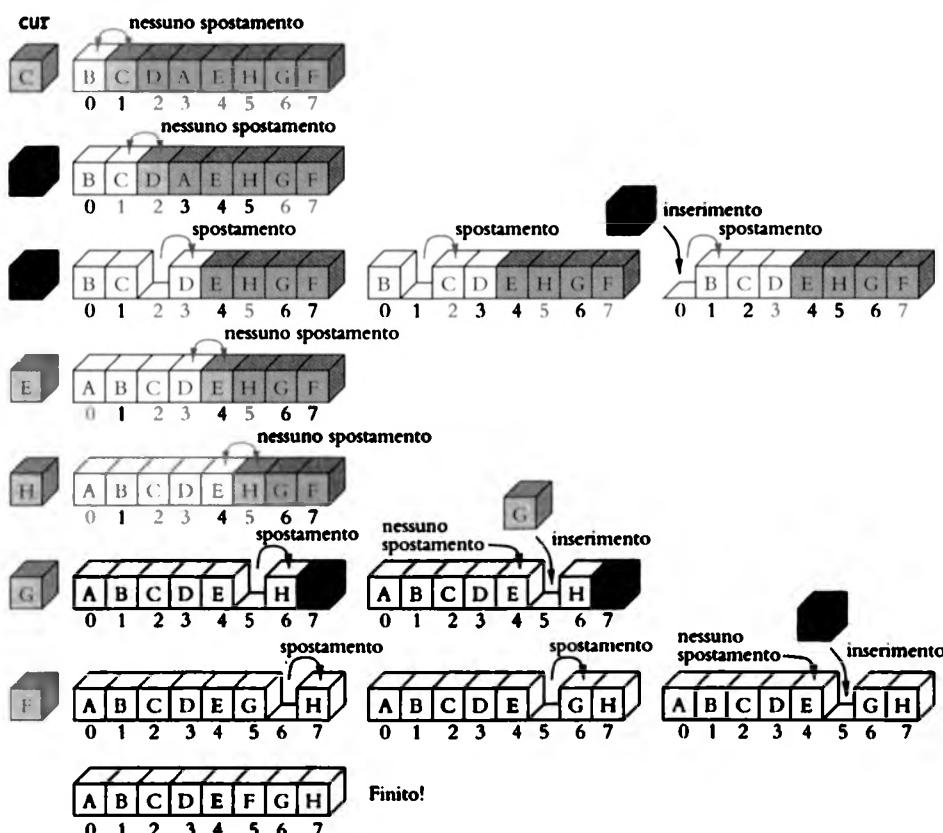


Figura 3.5: Esecuzione dell'algoritmo di ordinamento per inserimento su un array di otto caratteri. Ogni riga della figura corrisponde a un'iterazione del ciclo esterno, e ogni copia della sequenza presente su una riga corrisponde a un'iterazione del ciclo interno. L'elemento in esame, che si sta per inserire, è rappresentato in grigio chiaro ed è il valore di cur .

Codice 3.6: Il codice Java per eseguire l'ordinamento per inserimento su un array di caratteri.

```

1  /** Ordinamento non decrescente di un array di caratteri per inserimento */
2  public static void insertionSort(char[] data) {
3      int n = data.length;
4      for (int k = 1; k < n; k++) {           // inizia dal secondo carattere
5          char cur = data[k];                // è il momento di inserire cur=data[k]
6          int j = k;                         // trova l'indice j corretto per cur
7          while (j > 0 && data[j-1] > cur) {   // quindi, data[j-1] deve scavalcare cur
8              data[j] = data[j-1];            // fai scorrere data[j-1] verso destra
9              j--;                          // e considera il j precedente per cur
10         }
11         data[j] = cur;                  // questo è il posto giusto per cur
12     }
13 }
```

Osserviamo che, se un array è inizialmente ordinato, il ciclo interno dell'ordinamento per inserimento effettua un solo confronto, determinando che non è necessario alcuno scambio, e restituisce il controllo al ciclo esterno. Se, invece, l'array è inizialmente molto disordinato, è possibile, ovviamente, che ci sia molto più lavoro da fare: in effetti, lo sforzo da compiere per l'ordinamento per inserimento sarà massimo proprio quando l'array da ordinare è ordinato in senso decrescente.

3.1.3 I metodi di `java.util` per gli array e i numeri casuali

Dato che gli array sono così importanti, la libreria di Java contiene una classe, `java.util.Arrays`, che funge da contenitore di molti metodi statici che eseguono elaborazioni di uso frequente che coinvolgono array. Nei capitoli successivi descriveremo gli algoritmi su cui si basano questi metodi, mentre, per il momento, vediamo soltanto una panoramica di quelli più usati (con una discussione ulteriore nel Paragrafo 3.5.1):

equals(*A, B*): Restituisce `true` se e solo se l'array *A* e l'array *B* sono uguali. Due array sono considerati uguali se hanno lo stesso numero di elementi e ogni coppia di elementi corrispondenti nei due array è costituita da due elementi uguali: in pratica, se *A* e *B* contengono gli stessi valori, nello stesso ordine.

fill(*A, x*): Memorizza il valore *x* in ogni cella dell'array *A*, a condizione che il tipo dell'array *A* lo consenta.

copyOf(*A, n*): Restituisce un nuovo array di dimensione *n* tale che i suoi primi *k* elementi siano stati copiati ordinatamente dai primi *k* elementi di *A*, essendo *k* = $\min\{n, A.length\}$. Se *n* > *A.length*, agli ultimi *n* - *A.length* elementi del nuovo array sarà assegnato il valore predefinito per il tipo di array, ad esempio zero per array di `int` e `null` per array di oggetti di qualsiasi tipo.

copyOfRange(*A, s, t*): Restituisce un nuovo array di dimensione *t* - *s* (con *s* < *t*) tale che i suoi elementi siano stati copiati ordinatamente dagli elementi di *A* che vanno da *A[s]* a *A[t-1]*; se *t* > *A.length*, si usa l'array *A* come se fosse stato esteso quanto serve e riempito con valori predefiniti, come nel metodo `copyOf()`.

- toString(*A*):** Restituisce un oggetto di tipo `String` che è una rappresentazione del contenuto dell'array *A*, che inizia con [e termina con], con i singoli elementi che vengono visualizzati all'interno di tali parentesi quadre con una virgola (seguita da uno spazio) che separa ciascuna coppia di elementi adiacenti. La rappresentazione sotto forma di stringa di ciascun elemento *A[i]* è ottenuta invocando `String.valueOf(A[i])`, che restituisce la stringa "null" se il riferimento passato è uguale a `null`, e il risultato dell'invocazione di *A[i].toString()* in tutti gli altri casi.
- sort(*A*):** Ordina l'array *A* in base all'ordinamento naturale dei suoi elementi, che devono essere confrontabili. Gli algoritmi di ordinamento saranno trattati principalmente nel Capitolo 12.
- binarySearch(*A*, *x*):** Cerca il valore *x* nell'array *ordinato A*; se lo trova restituisce il suo indice, altrimenti, se *y* è l'indice della posizione in cui andrebbe inserito *x* per preservare l'ordinamento dell'array, restituisce $-y - 1$ (quindi se l'elemento cercato non è presente restituisce sempre un numero negativo, mentre se è presente restituisce un numero non negativo). L'algoritmo di ricerca binaria è descritto nel Paragrafo 5.1.3.

Dal momento che questi metodi sono statici, li si invoca usando la classe per rendere completo il loro nome, senza alcun particolare esemplare della classe stessa. Ad esempio, se `data` fosse un array, lo potremmo ordinare scrivendo `java.util.Arrays.sort(data)`, oppure, più semplicemente `Arrays.sort(data)` qualora avessimo importato la classe `java.util.Arrays` (si veda il Paragrafo 1.8).

Generazione di numeri pseudocasuali

Un'altra caratteristica presente nella libreria di Java, che risulta spesso utile quando si compilano programmi che usano array, è la possibilità di generare numeri pseudocasuali, cioè numeri che sembrano casuali (ma non sono necessariamente dei numeri veramente casuali). In particolare, Java mette a disposizione una classe, `java.util.Random`, i cui esemplari sono *generatori di numeri pseudocasuali*, cioè oggetti che costruiscono una sequenza di numeri statisticamente casuali. Queste sequenze, però, non sono effettivamente casuali, perché è possibile prevedere quale sarà il numero successivo presente nella sequenza conoscendo l'elenco dei numeri precedenti. In effetti, un generatore di numeri pseudocasuali molto utilizzato calcola il numero successivo, `next`, a partire dal precedente, `cur`, secondo questa formula:

```
next = (a * cur + b) % n;
```

dove *a*, *b* e *n* sono numeri interi opportuni, e % è il consueto operatore modulo. Il metodo usato da `java.util.Random` è abbastanza simile a questo, con *n* = 2^{48} . Si può dimostrare che una tale sequenza è statisticamente uniforme, una proprietà solitamente adeguata per la maggior parte delle applicazioni che necessitano di numeri casuali, come i giochi. Per applicazioni, come quelle relative alla sicurezza, che richiedono sequenze casuali non predibibili, non si può usare questa formula: in teoria, si deve utilizzare un campione

acquisito da una sorgente veramente casuale, come può essere un segnale radio proveniente dallo spazio profondo.

Dato che il numero successivo emesso da un generatore pseudocasuale è determinato dai numeri precedenti, serve un valore da cui partire, che viene chiamato *seme* (*seed*): la sequenza di numeri generata a partire da un determinato seme sarà sempre la stessa. Il seme viene dato da un esemplare della classe `java.util.Random` può essere fornito con il costruttore o mediante il suo metodo `setSeed()`.

Un trucco molto diffuso per ottenere una sequenza diversa ad ogni esecuzione del programma consiste nell'utilizzo di un seme che sia sicuramente diverso ogni volta, ad esempio un intervallo di tempo determinato da due eventi (inizio e fine dell'intervallo) messi in moto dall'utente (ad esempio, la pressione di due tasti sulla tastiera) oppure l'ora al momento dell'esecuzione, misurata in millisecondi a partire da una data di riferimento, che è il primo gennaio del 1970 (un valore che viene restituito dal metodo `System.currentTimeMillis()`).

Tra i metodi della classe `java.util.Random` citiamo:

- `nextBoolean():` Restituisce il successivo valore di tipo `boolean`.
- `nextDouble():` Restituisce il successivo valore di tipo `double`, compreso tra 0.0 (incluso) e 1.0 (escluso).
- `nextInt():` Restituisce il successivo valore di tipo `int`.
- `nextInt(n):` Restituisce il successivo valore di tipo `int`, compreso tra 0 (incluso) e `n` (escluso).
- `setSeed(s):` Imposta al valore `long s` il seme di questo generatore di numeri pseudocasuali.

Un esempio chiarificatore

Vediamo, nel Codice 3.7, un esempio breve, ma completo.

Codice 3.7: Un esempio di utilizzo di alcuni metodi della classe `java.util.Arrays`.

```

1 import java.util.Arrays;
2 import java.util.Random;
3 /** Programma che illustra alcuni utilizzi degli array */
4 public class ArrayTest {
5     public static void main(String[] args) {
6         int data[] = new int[10];
7         Random rand = new Random(); // un generatore di numeri pseudocasuali
8         rand.setSeed(System.currentTimeMillis()); // usa come seme l'ora d'esecuzione
9         // riempie l'array con numeri pseudocasuali compresi tra 0 e 99, estremi inclusi
10        for (int i = 0; i < data.length; i++)
11            data[i] = rand.nextInt(100); // il successivo numero pseudocasuale
12        int[] orig = Arrays.copyOf(data, data.length); // fa una copia dell'array "data"
13        System.out.println("arrays equal before sort: " + Arrays.equals(data, orig));
14        Arrays.sort(data); // ordina l'array data (mentre orig non viene modificato)
15        System.out.println("arrays equal after sort: " + Arrays.equals(data, orig));
16        System.out.println("orig = " + Arrays.toString(orig));
17        System.out.println("data = " + Arrays.toString(data));
18    }
19 }
```

Ecco un esempio di esecuzione del programma:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [41, 38, 48, 12, 28, 46, 33, 19, 10, 58]
data = [10, 12, 19, 28, 33, 38, 41, 46, 48, 58]
```

Eseguendolo un'altra volta, potremmo ottenere:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [87, 49, 70, 2, 59, 37, 63, 37, 95, 1]
data = [1, 2, 37, 37, 49, 59, 63, 70, 87, 95]
```

Usando un generatore di numeri pseudocasuali per determinare i valori su cui opera il programma, ogni volta che lo eseguiamo otteniamo un problema da risolvere diverso: in effetti, questa caratteristica è proprio quella che rende utile i generatori di numeri pseudocasuali nella fase di collaudo del codice, in particolar modo quando si opera con array. Ciò nonostante, è evidente che non dovremmo mai usare collaudi casuali come sostituti di nostri ragionamenti relativi al codice, perché questo potrebbe escludere alcuni casi speciali importanti. Si noti, ad esempio, che esiste la possibilità, ancorché poco probabile, che gli array `orig` e `data` siano uguali anche prima che `data` venga ordinato: ciò avviene, ovviamente, quando `orig` è già ordinato. La probabilità di questo evento è inferiore a 1 su 3 milioni, per cui è improbabile che si verifichi eseguendo il programma poche migliaia di volte, ma, in ogni caso, dobbiamo tener presente che la cosa è possibile.

3.1.4 Una semplice crittografia che usa array di caratteri

Un'importante applicazione di array di caratteri e stringhe è la *crittografia*, che è la scienza dei messaggi segreti. In questo campo ci si occupa del processo di *cifratura* (*encryption*), nel quale un messaggio, chiamato *testo in chiaro* (*plaintext*), viene convertito in un messaggio cifrato, chiamato, appunto, *testo cifrato* (*ciphertext*). Allo stesso modo, la crittografia studia anche, corrispondentemente, il modo per effettuare la *decifrazione* (*decryption*), che trasforma nuovamente il testo cifrato nel testo in chiaro originario.

Si presume che il primo schema di cifratura sia stato il *cifrario di Cesare*, che prende il nome da Giulio Cesare, che lo usava per proteggere importanti messaggi militari (ovviamente tutti i messaggi di Cesare erano scritti in latino, cosa che li renderebbe già incomprensibili per la maggior parte di noi!). Il cfrario di Cesare è uno schema molto semplice e può essere utilizzato per rendere poco comprensibile un messaggio scritto in qualsiasi linguaggio che usi parole composte con lettere di un determinato alfabeto.

Il cfrario di Cesare prevede la sostituzione di ciascuna lettera del messaggio con la lettera che si trova più avanti nell'alfabeto di un certo numero prefissato di posizioni. Ad esempio, in un messaggio scritto in lingua inglese, potremmo sostituire ogni lettera A con D, ogni B con E, ogni C con F, e così via, facendo scorrere ogni lettera in avanti di tre posizioni all'interno dell'alfabeto e proseguendo con la stessa regola fino alla lettera W, che viene sostituita da Z. A questo punto, facciamo in modo che lo schema di sostituzione si *riavvolga* all'inizio dell'alfabeto (con un meccanismo detto *wrap around*), sostituendo X con A, Y con B e Z con C.

Conversioni tra stringhe e array di caratteri

Dato che le stringhe sono oggetti immutabili, non possiamo modificare direttamente una stringa per cifrarla: quindi, genereremo una nuova stringa. Una tecnica piuttosto comoda per effettuare trasformazioni tra stringhe consiste nella creazione di un array di caratteri equivalente, per modificarlo e, poi, costruire una (nuova) stringa basata su di esso.

Java consente di convertire stringhe in array di caratteri, e viceversa. Data una stringa *S*, usando il metodo *S.toCharArray()* possiamo creare un nuovo array di caratteri il cui contenuto corrisponda a quello della stringa *S*. Ad esempio, se *S* = "bird", il metodo restituisce l'array di caratteri *A* = {'b', 'i', 'r', 'd'}. Per la conversione opposta, esiste una forma del costruttore di String che riceve come parametro un array di caratteri. Ad esempio, usando l'array di caratteri *A* = {'b', 'i', 'r', 'd'}, la sintassi *new String(A)* costruisce la stringa "bird".

Usare array di caratteri come codici di sostituzione

Se numeriamo le lettere dell'alfabeto con gli indici di un array, ad esempio associando A e 0, B e 1, C e 2, e così via, possiamo rappresentare la regola di sostituzione del cifrario di Cesare mediante proprio un array di caratteri, *encoder*, in modo che la lettera A venga sostituita con *encoder[0]*, B con *encoder[1]*, e così via. Quindi, per trovare la lettera che sostituisce un determinato carattere secondo il cifrario di Cesare, dobbiamo per prima cosa mettere in corrispondenza ciascuna lettera, da A a Z, con un numero da 0 a 25. Fortunatamente, per farlo possiamo sfruttare il fatto che i caratteri, in Java, sono rappresentati secondo lo standard Unicode mediante un codice che è un numero intero, e i codici delle lettere maiuscole dell'alfabeto latino sono numeri consecutivi (per semplicità, occupiamoci della cifratura delle sole lettere maiuscole).

Java consente di fare "sottrazioni" tra caratteri, ottenendo come risultato un numero intero che è uguale alla loro distanza all'interno dello schema di codifica Unicode. Data una variabile *c* che contiene una lettera maiuscola, l'assegnazione *j = c - 'A'* genera l'indice *j* che ci serve. Come verifica, osserviamo che, se il carattere *c* è 'A', si ottiene correttamente *j* = 0, mentre se il carattere *c* è 'B', si ottiene, di nuovo correttamente, *j* = 1, e così via: in generale, il numero intero *j* che si ottiene come risultato di quella sottrazione può essere usato come indice nel nostro array di codifica pre-calcolato, come si può vedere nella Figura 3.6.

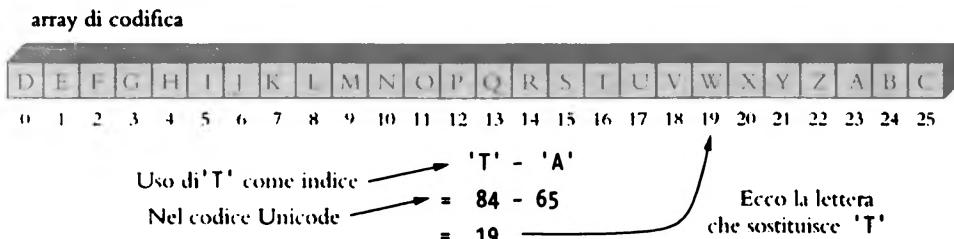


Figura 3.6: Uso di caratteri maiuscoli come indici, per realizzare la regola di sostituzione del cifrario di Cesare.

La procedura di *decifrazione* del messaggio si può implementare usando semplicemente un diverso array di caratteri per rappresentare la regola di sostituzione, un array il cui effetto sia quello di traslare i caratteri nella direzione opposta.

Nel Codice 3.8 presentiamo una classe Java che esegue la cifratura di Cesare con qualsiasi fattore di rotazione. Il costruttore della classe genera gli array necessari allo scorrimento per la cifratura e la decifratura, sulla base del fattore di rotazione ricevuto. Per fare questo sfruttiamo pesantemente l'operatore modulo, perché la cifratura di Cesare con il fattore r codifica la lettera avente indice k con la lettera di indice $(k + r) \bmod 26$ (essendo mod l'operatore modulo, che restituisce, come noto, il resto della divisione intera tra i suoi operandi). Questo operatore, in Java, si indica con il simbolo `%` ed è proprio quello che ci serve per eseguire facilmente l'operazione di *wrap around* al termine dell'alfabeto, già citata, perché $26 \bmod 26 = 0$, $27 \bmod 26 = 1$ e $28 \bmod 26 = 2$. L'array di decodifica per il cifrario di Cesare è esattamente l'inverso di quello usato per la cifratura: sostituiamo ciascuna lettera con quella che si trova r posizioni prima di essa, sempre usando la procedura di *wrap around*. Per evitare l'uso dell'operatore modulo con numeri negativi (che richiederebbe qualche approfondimento), in realtà sostituiamo la lettera avente indice k con la lettera di indice $(k - r + 26) \bmod 26$.

Disponendo degli array di codifica e di decodifica, gli algoritmi di cifratura e di decifrazione sono sostanzialmente identici tra loro, per cui li eseguiamo entrambi mediante un unico metodo privato ausiliario, che abbiamo chiamato `transform`. Questo metodo converte una stringa in un array di caratteri, esegue la traslazione visibile nella Figura 3.6 per ogni lettera maiuscola e, infine, restituisce una nuova stringa, costruita a partire dall'array così modificato.

Il metodo `main` della classe, come semplice collaudo, visualizza queste informazioni:

```
Encryption code = DEFGHIJKLMNOPQRSTUVWXYZABC
Decryption code = XYZABCDEFHIJKLMNOPQRSTUVWXYZ
Secret: WKH HDJOMH LV LQ SODB; PMWW DW MRH'V.
Message: THE EAGLE IS IN PLAY; MEET AT JOE'S.
```

Codice 3.8: Una classe completa che esegue il cifrario di Cesare.

```
1  /** Cifratura e decifrazione con il cifrario di Cesare. */
2  public class CaesarCipher {
3      protected char[] encoder = new char[26]; // array cifrante
4      protected char[] decoder = new char[26]; // array decifrante
5      /** Costruttore che inizializza l'array cifrante e l'array decifrante */
6      public CaesarCipher(int rotation) {
7          for (int k=0; k < 26; k++) {
8              encoder[k] = (char) ('A' + (k + rotation) % 26);
9              decoder[k] = (char) ('A' + (k - rotation + 26) % 26);
10         }
11     }
12     /** Restituisce il messaggio cifrato. */
13     public String encrypt(String message) {
14         return transform(message, encoder); // usa l'array cifrante
15     }
16     /** Restituisce il messaggio decifrato. */
17     public String decrypt(String secret) {
18         return transform(secret, decoder); // usa l'array decifrante
```

```

19 }
20 /** Restituisce la trasformazione della stringa ricevuta secondo il codice dato. */
21 private String transform(String original, char[] code) {
22     char[] msg = original.toCharArray();
23     for (int k=0; k < msg.length; k++) {
24         if (Character.isUpperCase(msg[k])) { // ecco una lettera da modificare
25             int j = msg[k] - 'A';           // sarà un valore da 0 a 25
26             msg[k] = code[j];            // sostituzione del carattere
27         }
28     return new String(msg);
29 }
30 /** Semplice metodo main per collaudare il cifrario di Cesare */
31 public static void main(String[] args) {
32     CaesarCipher cipher = new CaesarCipher(3);
33     System.out.println("Encryption code = " + new String(cipher.encoder));
34     System.out.println("Decryption code = " + new String(cipher.decoder));
35     String message = "THE EAGLE IS IN PLAY; MEET AT JOE'S.";
36     String coded = cipher.encrypt(message);
37     System.out.println("Secret: " + coded);
38     String answer = cipher.decrypt(coded);
39     System.out.println("Message: " + answer); // sarà di nuovo il testo in chiaro
40 }
41 }

```

3.1.5 Array bidimensionali e giochi posizionali

Molti giochi al calcolatore, tanto giochi di strategia quanto giochi di simulazione o giochi di combattimento in prima persona, usano oggetti che devono essere posizionati in uno spazio bidimensionale. Il software di questi *giochi posizionali* (*positional game*) necessita, appunto, di una modalità di rappresentazione di oggetti in uno spazio bidimensionale, cosa che si può ottenere in modo piuttosto naturale mediante un *array bidimensionale*, che usa due indici, diciamo *i* e *j*, per fare riferimento alle proprie celle. Solitamente il primo indice è il cosiddetto indice “di riga”, mentre il secondo è l’indice “di colonna”. Dato un tale array, possiamo gestire una scacchiera di gioco bidimensionale, così come effettuare altri tipi di elaborazioni che riguardino dati memorizzati in righe e colonne.

In Java gli array sono monodimensionali: usiamo un singolo indice per accedere a qualunque cella di un array. Ciò nonostante, è possibile definire anche in Java array bidimensionali: basta creare un array di array. In pratica, un array bidimensionale si definisce come un array in cui ciascuna cella contiene un riferimento a un altro array; tale array bidimensionale viene a volte chiamato anche *matrice* (*matrix*) e lo si dichiara in questo modo:

```
int[][] data = new int[8][10];
```

Questo enunciato crea un “array di array” bidimensionale, *data*, che è una matrice 8×10 , cioè ha 8 righe e 10 colonne. In effetti, *data* non è altro che un array di lunghezza 8 tale che ciascuno dei suoi elementi è un array di lunghezza 10 contenente numeri interi, come si può vedere nella Figura 3.7. Questi, quindi, sarebbero usi leciti dell’array *data*, posto che le variabili *i*, *j* e *k* siano dichiarate di tipo *int*:

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figura 3.7: Un array bidimensionale di numeri interi, data, avente 8 righe e 10 colonne. Il valore di data[3][5] è 100, mentre il valore di data[6][2] è 632.

```

data[i][i+1] = data[i][i] + 3;
j = data.length; // ora j vale 8
k = data[4].length; // ora k vale 10

```

Gli array bidimensionali sono utili in molte applicazioni, in particolare nell'analisi numerica, che fa un notevole uso di matrici. Invece di investigare in quella direzione, però, preferiamo presentare un'applicazione per un semplice gioco posizionale.

Il gioco Tic-Tac-Toe (*tris*)

Come sanno anche i bambini, il *Tic-Tac-Toe* (in italiano, *tris* o *filetto* o anche *schiera*) è un gioco per il quale si usa una scacchiera tre per tre. I due giocatori, chiamiamoli X e O, si alternano nel posizionare una delle proprie pedine in un riquadro della scacchiera, iniziando dal giocatore X. Se uno dei due riesce a piazzare tre delle proprie pedine sulla scacchiera in modo da formare una riga, una colonna o una delle due diagonali principali, vince il gioco.

È evidente che non si tratta di un gioco posizionale particolarmente sofisticato, e non è nemmeno molto divertente, perché il giocatore O, se è in gamba, è sempre in grado di ottenere un risultato di parità. Questo gioco, però, è un esempio molto semplice e comodo per illustrare l'utilizzo degli array bidimensionali nei giochi posizionali. Il software di molti altri giochi posizionali più complessi, come la dama o gli scacchi, oppure di giochi di simulazione molto diffusi è, in pratica, basato sullo stesso approccio che vedremo qui illustrato nell'utilizzo di un array bidimensionale per il Tic-Tac-Toe.

L'idea è quella di usare un array bidimensionale, *board*, per rappresentare la scacchiera del gioco. Le celle di questo array memorizzano valori che indicano se la corrispondente posizione della scacchiera è libera, oppure occupata da X o da O. In sostanza, *board* è una matrice tre per tre, la cui riga centrale, ad esempio, è composta dalle celle *board[1][0]*, *board[1][1]* e *board[1][2]*. Nel nostro caso, sceglieremo di usare celle che contengono numeri interi: 0 se la posizione è libera, 1 se contiene una X e -1 se contiene una O. Questa codifica ci consente di verificare in modo semplice se una determinata configurazione della scacchiera assegna la vittoria a X o a O: basta controllare se la somma dei valori di una riga, di una colonna o di una diagonale è uguale a 3 o, rispettivamente, a -3. Tutto quanto deciso è riassunto nella Figura 3.8.

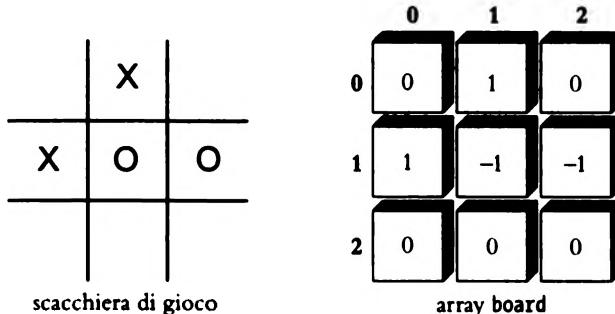


Figura 3.8: Una configurazione della scacchiera del gioco Tic-Tac-Toe e l'array bidimensionale di numeri interi, *board*, che la rappresenta.

Nel Codice 3.9 e 3.10 riportiamo il codice completo di una classe Java che gestisce la scacchiera del Tic-Tac-Toe per due giocatori, e nella Figura 3.9 riportiamo un esempio dell'esecuzione del programma. Il codice si occupa solamente di gestire la scacchiera e di registrare le mosse effettuate dai giocatori: non è in grado di giocare seguendo una strategia, quindi non consente di “giocare contro il computer”. La realizzazione di un tale programma andrebbe al di là degli obiettivi di questo capitolo, ma potrebbe essere senza alcun dubbio materia per un valido progetto in un corso di informatica (si veda l'Esercizio P-8.67).

Codice 3.9: Una classe, semplice ma completa, per gestire una partita di Tic-Tac-Toe tra due giocatori umani (continua nella sezione di Codice 3.10).

```

1  /** Simula una partita a Tic-Tac-Toe (non è in grado di giocare). */
2  public class TicTacToe {
3      public static final int X = 1, O = -1;           // giocatori
4      public static final int EMPTY = 0;                // posizione libera
5      private int board[][] = new int[3][3];          // scacchiera di gioco
6      private int player;                            // giocatore di turno
7      /** Costruttore */
8      public TicTacToe() { clearBoard(); }
9      /** Rende vuota la scacchiera */
10     public void clearBoard() {
11         for (int i = 0; i < 3; i++)
12             for (int j = 0; j < 3; j++)
13                 board[i][j] = EMPTY;           // ogni posizione viene resa libera
14             player = X;                  // il primo giocatore sarà 'X'
15     }
16     /** Mette una X o una 0 nella posizione i,j. */
17     public void putMark(int i, int j) throws IllegalArgumentException {
18         if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
19             throw new IllegalArgumentException("Invalid board position");
20         if (board[i][j] != EMPTY)
21             throw new IllegalArgumentException("Board position occupied");
22         board[i][j] = player; // memorizza il simbolo del giocatore di turno
23         player = -player;    // scambia i giocatori (sfruttando il fatto che 0 == - X)
24     }
25     /** Verifica se il giocatore indicato vince con l'attuale configurazione. */
26     public boolean isWin(int mark) {
27         return ((board[0][0] + board[0][1] + board[0][2] == mark*3) // riga 0

```

```

28     || (board[1][0] + board[1][1] + board[1][2] == mark*3) // riga 1
29     || (board[2][0] + board[2][1] + board[2][2] == mark*3) // riga 2
30     || (board[0][0] + board[1][0] + board[2][0] == mark*3) // colonna 0
31     || (board[0][1] + board[1][1] + board[2][1] == mark*3) // colonna 1
32     || (board[0][2] + board[1][2] + board[2][2] == mark*3) // colonna 2
33     || (board[0][0] + board[1][1] + board[2][2] == mark*3) // diagonale
34     || (board[2][0] + board[1][1] + board[0][2] == mark*3)); // altra diagonale
35   }
36   /** Restituisce il codice del vincitore o 0 se parità o partita non finita.*/
37   public int winner() {
38     if (isWin(X))
39       return(X);
40     else if (isWin(0))
41       return(0);
42     else
43       return(0);
44   }

```

Codice 3.10: Una classe, semplice ma completa, per gestire una partita di Tic-Tac-Toe tra due giocatori umani (prosegue dalla sezione di Codice 3.9).

```

45   /** Restituisce una stringa che descrive la configurazione della scacchiera. */
46   public String toString() {
47     StringBuilder sb = new StringBuilder();
48     for (int i=0; i<3; i++) {
49       for (int j=0; j<3; j++) {
50         switch (board[i][j]) {
51           case X:      sb.append("X"); break;
52           case O:      sb.append("O"); break;
53           case EMPTY: sb.append(" "); break;
54         }
55         if (j < 2) sb.append("|");          // fine della riga
56       }
57       if (i < 2) sb.append("\n----\n");    // fine della colonna
58     }
59     return sb.toString();
60   }
61   /** Gioca una partita come collaudo */
62   public static void main(String[] args) {
63     TicTacToe game = new TicTacToe();
64     /* mosse di X: */           /* mosse di O: */
65     game.putMark(1,1);          game.putMark(0,2);
66     game.putMark(2,2);          game.putMark(0,0);
67     game.putMark(0,1);          game.putMark(2,1);
68     game.putMark(1,2);          game.putMark(1,0);
69     game.putMark(2,0);
70     System.out.println(game);
71     int winningPlayer = game.winner();
72     String[] outcome = {"O wins", "Tie", "X wins"}; // sfrutta l'ordinamento
73     System.out.println(outcome[1 + winningPlayer]);
74   }
75 }

```

```

0|X|0
-----
0|X|X
-----
X|0|X
Tie

```

Figura 3.9: Visualizzazione prodotta dall'esecuzione della classe TicTacToe.

3.2 Liste semplicemente concatenate

Nel paragrafo precedente abbiamo presentato gli array come strutture dati e abbiamo analizzato alcune applicazioni che li usano. Gli array sono una buona soluzione quando si vogliono memorizzare dati in un determinato ordine, ma presentano alcuni svantaggi: la capacità dell'array deve essere fissata nel momento in cui lo si crea; inoltre, gli inserimenti e le rimozioni di dati in posizioni interne all'array possono richiedere molto tempo, se molti sono gli elementi che si devono far scorrere.

In questo paragrafo presentiamo una struttura dati nota come *lista concatenata (linked list)*, che costituisce un'alternativa alle strutture basate su array. Una lista concatenata, nella sua forma più semplice, è costituita da *nodi* che, insieme, formano una sequenza lineare. In una *lista semplicemente concatenata (singly linked list)*, detta anche *lista a concatenazione singola o semplice* ciascun nodo memorizza un riferimento a un oggetto che è un elemento della sequenza, oltre a un riferimento al nodo successivo della lista, come si può vedere nella Figura 3.10.

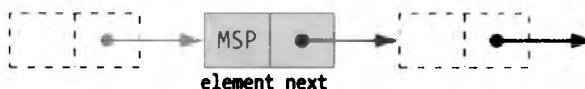


Figura 3.10: Esempio di un esemplare di nodo che fa parte di una lista semplicemente concatenata. Il campo *element* del nodo fa riferimento a un oggetto che è uno degli elementi della sequenza (in questo esempio, il codice aeroportuale MSP), mentre il campo *next* si riferisce al nodo successivo della lista (oppure vale *null* se non c'è un altro nodo).

La rappresentazione di una lista concatenata si basa sulla cooperazione di molti oggetti, come evidenziato nella Figura 3.11. Come minimo, un esemplare di lista concatenata deve conservare un riferimento al primo nodo della lista, noto come *testa (head)*: senza un riferimento esplicito al nodo iniziale della lista, non ci sarebbe alcun modo per rintracciarlo e, conseguentemente, non si potrebbe accedere a nessun altro nodo. L'ultimo nodo della lista prende il nome di *coda (tail)*. La coda di una lista può essere trovata *scandendo (traversing)* l'intera lista concatenata: partendo dalla testa e spostandosi da un nodo al successivo seguendo, in ciascun nodo, il suo riferimento *next*. L'ultimo nodo, la coda, può essere identificato facilmente, in quanto è l'unico ad avere *null* memorizzato nel campo *next*. Questa procedura di scansione della lista viene chiamata *salto dei collegamenti (link hopping)* o *dei puntatori (pointer hopping)*. A dispetto di tale possibilità e a favore di una maggiore efficienza, solitamente si

memorizza un riferimento esplicito al nodo che costituisce la coda della lista, in modo da evitare la scansione completa. Analogamente e per simili motivi di efficienza, è decisamente frequente che un esemplare di lista concatenata memorizzi il valore del numero totale dei propri nodi (detto anche *dimensione*, *size*, della lista), evitando così di dover scandire tutta la lista al solo scopo di contare quanti nodi vi sono presenti.

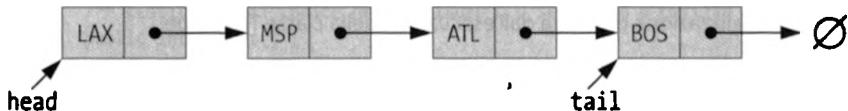


Figura 3.11: Esempio di lista semplicemente concatenata i cui elementi sono stringhe che rappresentano codici aeroportuali. L'esemplare di lista contiene un membro di nome *head*, che fa riferimento al primo nodo della lista, e un altro membro, di nome *tail*, che si riferisce all'ultimo nodo della lista stessa. Il valore *null* è stato qui rappresentato dal simbolo dell'insieme vuoto, \emptyset .

Inserire un elemento all'inizio di una lista semplicemente concatenata

Un'importante proprietà delle liste concatenate è la mancanza di una dimensione predeterminata: la struttura usa uno spazio proporzionale al suo numero di elementi. Quando usiamo una lista semplicemente concatenata, possiamo facilmente inserire un nuovo elemento all'inizio della lista, cioè in corrispondenza della sua testa, mediante la procedura che si può vedere nella Figura 3.12 e che è descritta dallo pseudocodice riportato nel Codice 3.11. L'idea è sostanzialmente quella di creare un nuovo nodo, assegnare al suo campo *element* un riferimento al nuovo elemento da inserire e al suo campo *next* l'attuale valore di *head*, per poi assegnare a *head* un riferimento al nuovo nodo appena creato.

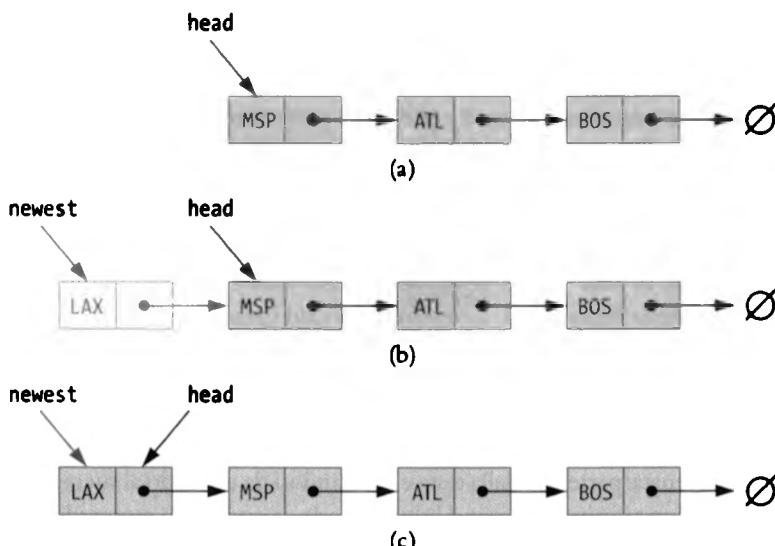


Figura 3.12: Inserimento di un nuovo elemento all'inizio di una lista semplicemente concatenata: (a) prima dell'inserimento; (b) dopo la creazione del nuovo nodo e il suo collegamento all'inizio della lista; (c) dopo l'aggiornamento del riferimento *head* in modo che punti al nuovo nodo.

Codice 3.11: Inserimento di un nuovo elemento all'inizio di una lista semplicemente concatenata. Si noti come al campo next del nuovo nodo venga assegnato un valore *prima* che il valore di head venga modificato per puntare proprio a tale nuovo nodo. Se la lista era inizialmente vuota (per cui il valore di head era null), conseguentemente il valore del campo next del nuovo nodo diventa null.

Algoritmo addFirst(*e*):

```
newest = Node(e)           { crea un nuovo esemplare di nodo il cui element punta a e }
newest.next = head           { assegna al campo next del nuovo nodo il valore attuale di head }
head = newest               { assegna alla variabile head il riferimento al nuovo nodo }
size = size + 1             { incrementa il numero di nodi della lista }
```

Inserire un elemento alla fine di una lista semplicemente concatenata

Come si può vedere nella Figura 3.13, è altrettanto semplice inserire un elemento alla fine di una lista concatenata, a patto che si sia tenuta traccia del riferimento tail all'ultimo nodo. In questo caso, creiamo un nuovo nodo, assegniamo il valore null al suo campo next e facciamo in modo che il campo next del nodo a cui punta tail faccia riferimento a questo nuovo nodo, infine aggiorniamo la variabile tail stessa, facendola puntare al nuovo nodo. Il Codice 3.12 riporta il corrispondente pseudocodice.

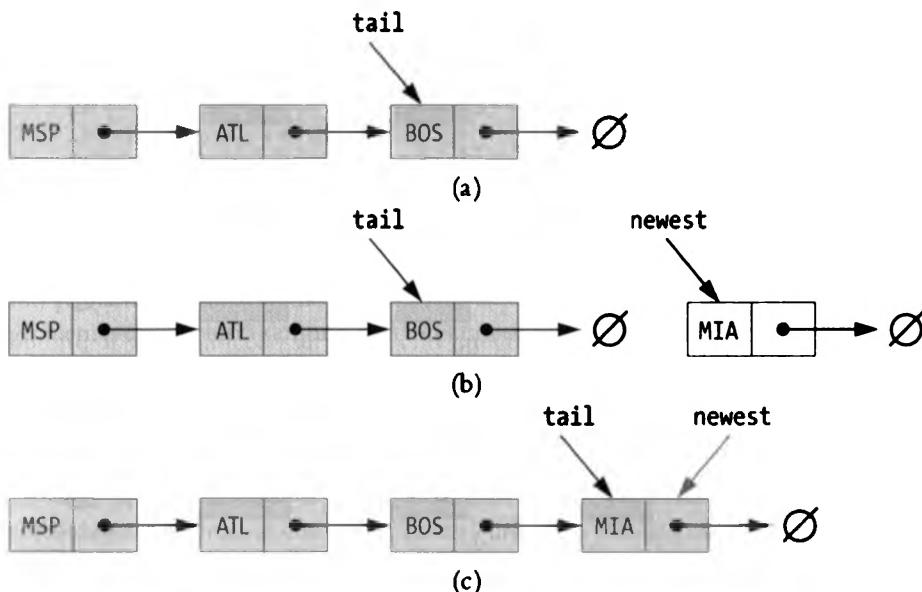


Figura 3.13: Inserimento di un nuovo elemento alla fine di una lista semplicemente concatenata: (a) prima dell'inserimento; (b) dopo la creazione del nuovo nodo; (c) dopo l'aggiornamento del riferimento tail. Si noti che, passando da (b) a (c), bisogna prima aggiornare il campo next del nodo a cui fa riferimento tail e solo successivamente aggiornare la variabile tail in modo che punti al nuovo nodo.

Codice 3.12: Inserimento di un nuovo elemento alla fine di una lista semplicemente concatenata. Si noti come al campo `next` del nodo a cui fa riferimento il vecchio valore di `tail` venga assegnato un valore *prima* che il valore di `tail` venga modificato per puntare proprio a tale nuovo nodo. Questo codice necessita di qualche modifica per poter gestire l'inserimento in una lista vuota, perché in tal caso, ovviamente, `tail` non farebbe inizialmente riferimento ad alcun nodo.

Algoritmo `addLast(e)`:

```

newest = Node(e)           { crea un nuovo esemplare di nodo il cui element punta a e }
newest.next = null         { assegna al campo next del nuovo nodo il valore null }
tail.next = newest         { assegna al next del vecchio tail il riferimento al nuovo nodo }
tail = newest              { assegna alla variabile tail il riferimento al nuovo nodo }
size = size + 1            { incrementa il numero di nodi della lista }

```

Eliminare un elemento da una lista semplicemente concatenata

L'eliminazione di un elemento dall'*inizio* di una lista semplicemente concatenata è essenzialmente l'operazione inversa dell'inserimento di un nuovo elemento a quella stessa estremità. Tale eliminazione è illustrata nella Figura 3.14 e descritta in dettaglio nel Codice 3.13.

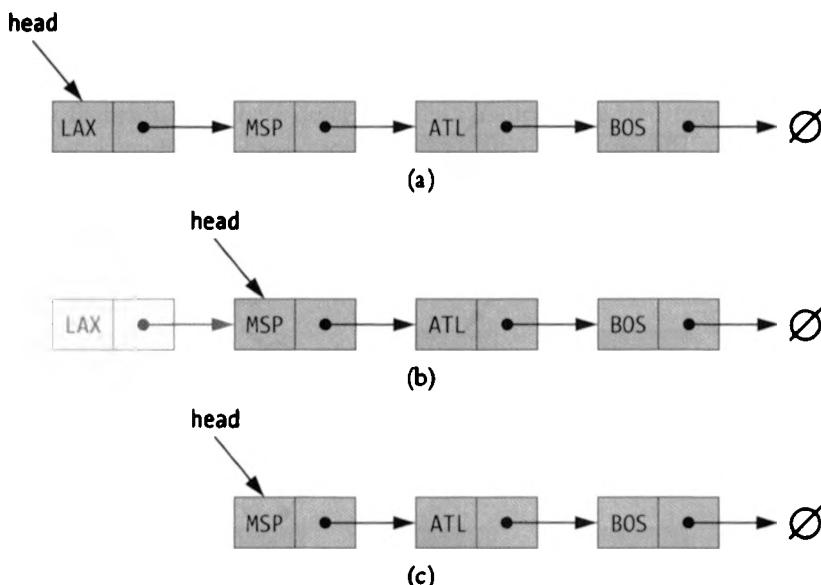


Figura 3.14: Eliminazione di un elemento dall'inizio di una lista semplicemente concatenata: (a) prima dell'eliminazione; (b) dopo lo "scollegamento" del vecchio nodo iniziale; (c) situazione finale.

Codice 3.13: Eliminazione del nodo iniziale di una lista semplicemente concatenata.

Algoritmo `removeFirst(e)`:

```

if head == null then
    la lista è vuota
head = head.next          { fa in modo che head punti al nodo successivo (o valga null) }
size = size - 1            { decrementa il numero di nodi della lista }

```

Mortunatamente non è altrettanto semplice eliminare l'ultimo nodo di una lista semplicemente concatenata. Anche se si memorizza nella variabile `tail` il riferimento diretto all'ultimo nodo della lista, si deve comunque accedere al nodo *che lo precede* per poter eliminare *l'ultimo* nodo, ma questa operazione non può essere compiuta seguendo banalmente un riferimento a partire dall'ultimo nodo: l'unico modo per accedere al penultimo nodo è la transizione della lista a partire dal suo primo nodo (a cui si accede tramite `head`), seguendo poi, uno dopo l'altro, i riferimenti `next`. Purtroppo questa serie di operazioni di *link hopping* può richiedere molto tempo: se vogliamo realizzare in modo efficiente questa operazione (cioè l'eliminazione dell'ultimo nodo), dovremo usare una *lista doppiamente concatenata* (*doubly linked list*, o *lista a concatenazione doppia*), come vedremo nel Paragrafo 3.4.

3.2.1 Realizzare una lista semplicemente concatenata

In questo paragrafo vedremo una realizzazione concreta della classe `SinglyLinkedList`, i cui esemplari sono liste semplicemente concatenate e mettono a disposizione i seguenti metodi:

- `size()`: Restituisce il numero di elementi presenti nella lista.
- `isEmpty()`: Restituisce `true` se e solo se la lista è vuota.
- `first()`: Restituisce il primo elemento della lista (senza eliminarlo).
- `last()`: Restituisce l'ultimo elemento della lista (senza eliminarlo).
- `addFirst(e)`: Aggiunge un nuovo elemento (`e`) all'inizio della lista.
- `addLast(e)`: Aggiunge un nuovo elemento (`e`) alla fine della lista.
- `removeFirst()`: Elimina dalla lista il suo primo elemento e lo restituisce.

Se i metodi `first()`, `last()` o `removeFirst()` vengono invocati con una lista vuota, restituiscono semplicemente un riferimento `null`, lasciando la lista immutata.

Dato che non ci interessa il tipo degli elementi che vengono memorizzati nella lista, useremo l'infrastruttura Java per la progettazione di strutture generiche (vista nel Paragrafo 2.5.2) per definire la nostra classe, usando il parametro formale di tipo `E` per indicare il tipo degli elementi, che verrà definito dall'utilizzatore della lista.

La nostra implementazione sfrutta anche i vantaggi derivanti dal fatto che Java consente la definizione di *classi annidate* (si veda il Paragrafo 2.6): definiamo la classe privata `Node` all'interno dell'ambito di visibilità relativo alla classe `SinglyLinkedList`. Il Codice 3.14 contiene la definizione della classe `Node`, mentre il Codice 3.15 presenta la parte rimanente della classe `SinglyLinkedList`. Il fatto che `Node` sia una classe annidata rende più stringente l'incapsulamento, evitando che gli utilizzatori della lista concatenata debbano preoccuparsi dei dettagli realizzativi di nodi e collegamenti. Questo schema progettuale consente al compilatore Java di distinguere tra questo tipo di nodi e altre forme di nodi che potremmo definire al servizio di altre strutture.

Codice 3.14: La classe `Node`, annidata nella classe `SinglyLinkedList` (la parte rimanente della classe `SinglyLinkedList` si trova nel Codice 3.15).

```

1  public class SinglyLinkedList<E> {
2      //----- classe annidata Node -----
3      private static class Node<E> {
4          private E element;    // riferimento all'elemento memorizzato in questo nodo
5          private Node<E> next; // riferimento al nodo successivo nella lista
6          public Node(E e, Node<E> n) {

```

```

7     element = e;
8     next = n;
9 }
10    public E getElement() { return element; }
11    public Node<E> getNext() { return next; }
12    public void setNext(Node<E> n) { next = n; }
13 } //----- fine della classe annidata Node -----
... seguirà il resto della classe SinglyLinkedList ...

```

Codice 3.15: Definizione della classe SinglyLinkedList (da mettere assieme al codice della classe annidata Node, nella sezione Codice 3.14).

```

1  public class SinglyLinkedList<E> {
...
    (qui ci vuole il codice della classe annidata Node)

14   // variabili di esemplare di SinglyLinkedList
15   private Node<E> head = null; // nodo iniziale della lista (o null se è vuota)
16   private Node<E> tail = null; // nodo finale della lista (o null se è vuota)
17   private int size = 0; // numero di nodi della lista
18   public SinglyLinkedList() { } // costruisce una lista inizialmente vuota
19   // metodi di accesso
20   public int size() { return size; }
21   public boolean isEmpty() { return size == 0; }
22   public E first() { // restituisce il primo elemento senza eliminarlo
23       if (isEmpty()) return null;
24       return head.getElement();
25   }
26   public E last() { // restituisce l'ultimo elemento senza eliminarlo
27       if (isEmpty()) return null;
28       return tail.getElement();
29   }
30   // metodi di aggiornamento
31   public void addFirst(E e) { // aggiunge l'elemento e all'inizio della lista
32       head = new Node<E>(e, head); // crea un nuovo nodo e lo collega alla lista
33       if (size == 0)
34           tail = head; // caso speciale: il nuovo nodo diventa anche tail
35       size++;
36   }
37   public void addLast(E e) { // aggiunge l'elemento e alla fine della lista
38       Node<E> newest = new Node<E>(e, null); // il nodo che diventerà l'ultimo
39       if (isEmpty())
40           head = newest; // caso speciale: lista inizialmente vuota
41       else
42           tail.setNext(newest); // il nuovo nodo diventa il successivo di tail
43           tail = newest; // il nuovo nodo diventa l'ultimo
44           size++;
45   }
46   public E removeFirst() { // elimina il primo elemento e lo restituisce
47       if (isEmpty()) return null; // non c'è niente da eliminare
48       E answer = head.getElement();
49       head = head.getNext(); // diventa null se la lista ha un solo nodo
50       size--;
51       if (size == 0)
52           tail = null; // caso speciale: ora la lista è vuota
53       return answer;
54   }
55 }

```

3.3 Liste concatenate circolari

Tradizionalmente ci si immagina che le liste concatenate memorizzino una sequenza di dati secondo uno schema lineare, dal primo all'ultimo, ma ci sono molte applicazioni in cui i dati vengono considerati in modo più naturale come aventi una *disposizione ciclica*, con relazioni di prossimità ben definite ma senza un inizio o una fine.

Ad esempio, molti giochi con più giocatori sono basati su turni di gioco, con il giocatore A che gioca per primo, poi passa la mano al giocatore B, quindi a C e così via, tornando prima o poi al giocatore A, quindi di nuovo a B, ripetendo lo schema di gioco. Un altro esempio: spesso, in città, i treni della metropolitana viaggiano a ciclo continuo, rispettando le fermate nell'ordine previsto, ma senza che esista una vera e propria prima (o ultima) fermata. Vedremo ora un altro importante esempio di disposizione ciclica che ha interesse nel contesto dei sistemi operativi per calcolatore.

3.3.1 Pianificazione circolare (*round robin*)

Uno dei compiti principali di un sistema operativo è la gestione dei molti processi solitamente in esecuzione in un calcolatore, che comprendono anche i processi attivi su una o più CPU (*central processing unit*, unità centrale di elaborazione). Per consentire a un numero arbitrario di processi di apparire attivi e di rispondere alle richieste dell'utente, la maggior parte dei sistemi operativi consente effettivamente la condivisione della CPU tra i processi, usando, sotto diverse forme, un algoritmo che prende il nome di *round-robin scheduling* (*pianificazione circolare*). Viene assegnato a un processo un breve intervallo di tempo (noto come *time slice* o *porzione di tempo*) nel quale utilizzare la CPU, interrompendosi al termine della porzione di tempo assegnata, anche se non ha portato a termine il proprio lavoro, per passare a un nuovo processo. A ogni processo viene assegnata la propria porzione di tempo, secondo uno schema circolare. I nuovi processi messi in esecuzione sul calcolatore entrano nel sistema di gestione, mentre quelli che terminano il proprio lavoro ne escono.

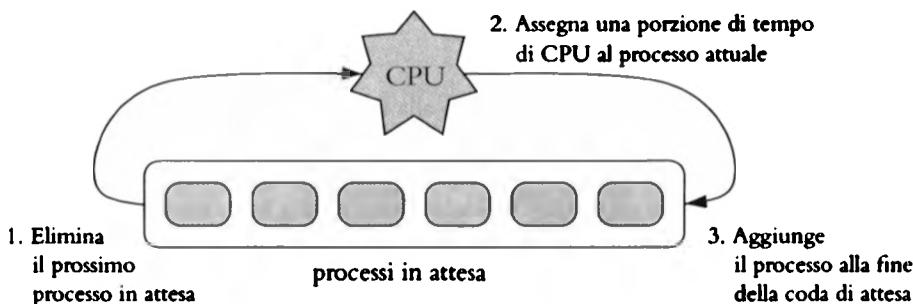


Figura 3.15: Le tre fasi, eseguite ripetutamente, di una pianificazione che segue uno schema *round-robin*.

Un pianificatore che segua uno schema *round-robin* potrebbe essere realizzato mediante una lista concatenata tradizionale, L , eseguendo ripetutamente le seguenti fasi (Figura 3.15):

1. Elabora $p = L.\text{removeFirst}()$
2. Assegna una porzione di tempo al processo p
3. $L.\text{addLast}(p)$

Sfortunatamente, l'uso di una lista concatenata tradizionale per risolvere questo problema ha alcuni svantaggi. Eliminare ripetutamente un nodo da un'estremità della lista, soltanto per creare poi un nuovo nodo contenente lo stesso elemento e reinserirlo nella lista, è inutilmente inefficiente, per non parlare dei vari aggiornamenti che si rendono necessari per decrementare e incrementare la dimensione della lista e per scollare e ricollegare i nodi.

Nel seguito di questo paragrafo dimostreremo come si possa apportare una piccola modifica a una lista semplicemente concatenata per ottenere una struttura dati più efficiente per rappresentare una disposizione circolare di elementi.

3.3.2 Progettare e realizzare una lista concatenata circolare

In questo paragrafo presentiamo il progetto di una struttura nota come *lista concatenata circolare* (*circularly linked list*), che è essenzialmente una lista semplicemente concatenata in cui il campo `next` dell'ultimo nodo punta (“all’indietro”) al primo nodo della lista (mentre normalmente ha il valore `null`), come mostrato nella Figura 3.16.

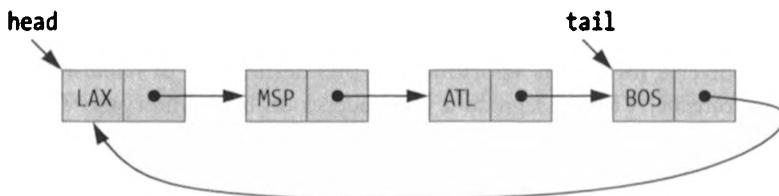


Figura 3.16: Esempio di una lista semplicemente concatenata con una struttura circolare.

Usiamo questo modello per progettare e realizzare una nuova classe, `CircularlyLinkedList`, che fornisce supporto a tutti i comportamenti pubblici della nostra classe `SinglyLinkedList`, aggiungendo un ulteriore metodo:

`rotate():` Sposta il primo elemento alla fine della lista.

Con questa nuova operazione, la pianificazione circolare (*round robin*) può essere implementata in modo efficiente eseguendo ripetutamente su una lista concatenata circolare C le fasi seguenti:

1. Assegna una porzione di tempo al processo $C.\text{first}()$
2. $C.\text{rotate}()$

Un’ulteriore ottimizzazione

Nel progettare questa nuova classe, implementiamo un’ulteriore ottimizzazione: non abbiamo più bisogno di gestire esplicitamente il riferimento `head`, è sufficiente memorizzare il riferimento `tail`, con il quale possiamo raggiungere il primo nodo seguendo il riferimento `tail.getNext()`. Gestire il solo riferimento `tail` ci consente non soltanto di risparmiare un

po' di memoria, ma anche di rendere il codice più semplice e più efficiente, perché non si rende più necessario tenere aggiornato il riferimento `head`. In effetti, questa nostra nuova implementazione è decisamente migliore di quella originale, anche se il nuovo metodo `rotate` non dovesse servire.

Operazioni su una lista concatenata circolare

L'implementazione del nuovo metodo `rotate` è abbastanza banale. Non spostiamo nodi né elementi, ma facciamo semplicemente "avanzare" il riferimento `tail` in modo che punti al nodo che segue `tail` (cioè a quello che è implicitamente il nodo iniziale della lista). La Figura 3.17 illustra il funzionamento di questa operazione usando una visualizzazione più simmetrica della lista concatenata circolare.

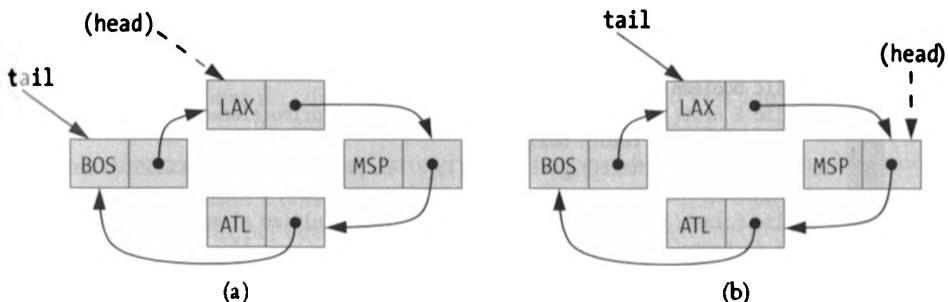


Figura 3.17: L'operazione di rotazione in una lista concatenata circolare: (a) prima della rotazione, la lista rappresenta la sequenza { LAX, MSP, ATL, BOS }; (b) dopo la rotazione, la lista rappresenta la sequenza { MSP, ATL, BOS, LAX }. Abbiamo visualizzato il riferimento隐式的 `head` (indicato tra parentesi), che nella nuova implementazione viene individuato come `tail.getNext()`.

Possiamo aggiungere un nuovo elemento all'inizio della lista creando un nuovo nodo e collegandolo subito *dopo* la fine della lista, come si può vedere nella Figura 3.18. Per implementare il metodo `addLast`, invece, possiamo usare un'invocazione del metodo `addFirst`, facendo poi immediatamente avanzare il riferimento `tail` in modo che il nodo appena aggiunto diventi l'ultimo.

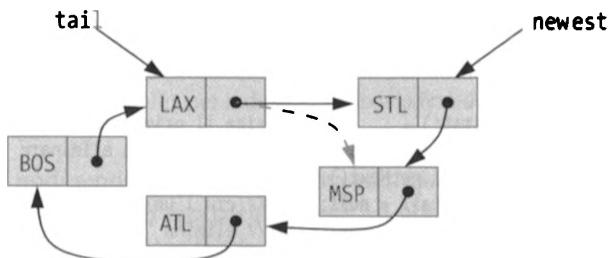


Figura 3.18: Effetto dell'invocazione di `addFirst(STL)` sulla lista concatenata circolare della Figura 3.17(b). La variabile `newest` è locale ed esiste soltanto durante l'esecuzione del metodo. Si noti che, quando l'operazione è stata completata, `STL` è il primo elemento della lista, dal momento che è memorizzato all'interno del suo nodo iniziale, raggiungibile usando il riferimento `tail.getNext()`.

L'eliminazione del primo nodo di una lista concatenata circolare può essere ottenuta semplicemente aggiornando il campo `next` del nodo a cui punta `tail`, in modo che scavalchi il primo nodo. Nel Codice 3.16 forniamo l'implementazione di tutti i metodi della classe `CircularlyLinkedList`.

Codice 3.16: Implementazione della classe `CircularlyLinkedList`.

```

1  public class CircularlyLinkedList<E> {
...
    (qui ci vuole la classe annidata Node, identica a quella di SinglyLinkedList)

14   // variabili di esemplare di CircularlyLinkedList
15   private Node<E> tail = null; // nodo finale della lista (non usiamo head)
16   private int size = 0; // numero di nodi della lista
17   public CircularlyLinkedList() { } // costruisce una lista inizialmente vuota
18   // metodi di accesso
19   public int size() { return size; }
20   public boolean isEmpty() { return size == 0; }
21   public E first() { // restituisce il primo elemento senza eliminarlo
22       if (isEmpty()) return null;
23       return tail.getNext().getElement(); // il primo nodo è IL SUCCESSIVO dell'ultimo
24   }
25   public E last() { // restituisce l'ultimo elemento senza eliminarlo
26       if (isEmpty()) return null;
27       return tail.getElement();
28   }
29   // metodi di aggiornamento
30   public void rotate() // porta il primo elemento in fondo alla lista
31       if (tail != null) // se la lista è vuota, non fa niente
32           tail = tail.getNext(); // il primo nodo diventa l'ultimo
33   }
34   public void addFirst(E e) { // aggiunge l'elemento e all'inizio della lista
35       if (size == 0) {
36           tail = new Node<E>(e, null);
37           tail.setNext(tail); // fa riferimento circolarmente a se stesso
38       } else {
39           Node<E> newest = new Node<E>(e, tail.getNext());
40           tail.setNext(newest);
41       }
42       size++;
43   }
44   public void addLast(E e) { // aggiunge l'elemento e alla fine della lista
45       addFirst(e); // inserisce il nuovo elemento all'inizio
46       tail = tail.getNext(); // ora il nuovo elemento è diventato l'ultimo
47   }
48   public E removeFirst() { // elimina il primo elemento e lo restituisce
49       if (isEmpty()) return null; // non c'è niente da eliminare
50       Node<E> head = tail.getNext();
51       if (head == tail) tail = null; // era l'unico nodo della lista
52       else tail.setNext(head.getNext()); // elimina "head" dalla lista
53       size--;
54       return head.getElement();
55   }
56 }
```

3.4 Liste doppiamente concatenate

In una lista semplicemente concatenata ogni nodo ha un riferimento al nodo che lo segue nella sequenza e l'utilità di una tale rappresentazione per la gestione di una sequenza di elementi è stata ampiamente dimostrata nei paragrafi precedenti. Ci sono, però, alcune limitazioni che conseguono dall'asimmetria di una tale lista semplicemente concatenata. Nel Paragrafo 3.2 abbiamo visto che possiamo inserire in modo efficiente un nuovo nodo tanto all'inizio quanto alla fine di una lista semplicemente concatenata, e in modo altrettanto efficiente possiamo eliminare il suo elemento iniziale, ma non siamo in grado di eliminare in modo analogo il suo ultimo elemento. Più in generale, non siamo in grado di eliminare in modo efficiente un nodo qualsiasi da una posizione interna alla lista se disponiamo solamente del riferimento a tale nodo, perché non siamo in grado di individuare rapidamente il nodo che *precede* quello da eliminare (e proprio in tale nodo dobbiamo aggiornare il campo *next*).

Per migliorare la simmetria della struttura, definiamo una lista concatenata nella quale ciascun nodo memorizza un riferimento esplicito al nodo che lo precede e al nodo che lo segue nella sequenza: quella che si chiama *lista doppiamente concatenata* (*doubly linked list*, o *lista a concatenazione doppia*). In queste liste aumenta il numero di operazioni di aggiornamento che sono $O(1)$, arrivando a comprendere inserimenti e rimozioni in qualunque posizione della lista. Continuiamo, per coerenza, a chiamare *next* (*successivo*) il riferimento al nodo che ne segue un altro, mentre useremo il nome *prev* (che sta per *previous*, cioè *precedente*) per il riferimento al nodo che ne precede un altro.

Sentinella iniziale e finale

Per evitare di dover gestire alcuni casi speciali quando si opera nei pressi delle estremità di una lista doppiamente concatenata, può essere utile aggiungere un nodo speciale all'inizio e alla fine della lista stessa: un nodo *header* (*intestazione* o *iniziale*) all'inizio e un nodo *trailer* (*terminale* o *appendice*) alla fine della lista. Questi nodi "inutili" vengono chiamati *sentinelle* e non memorizzano elementi della sequenza. La Figura 3.19 mostra una lista doppiamente concatenata dotata di tali sentinelle.

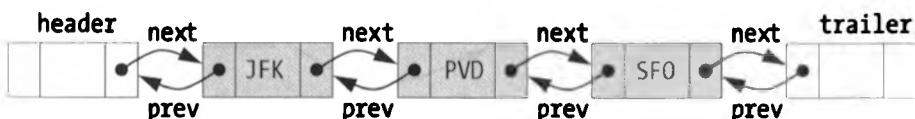


Figura 3.19: Una lista doppiamente concatenata che rappresenta la sequenza { JFK, PVD, SFO }, usando le sentinelle header e trailer per indicare l'inizio e la fine della lista.

Quando si usano i nodi sentinella, una lista vuota viene inizializzata in modo che il campo *next* del nodo a cui punta *header* faccia riferimento al nodo a cui punta *trailer*, e il campo *prev* del nodo a cui punta *trailer* faccia riferimento al nodo a cui punta *header*; gli altri campi dei nodi sentinella sono, invece, ininfluenti (presumibilmente, in Java, avranno il valore `null`). Se la lista non è vuota, il campo *next* del nodo *header* farà riferimento a un nodo che contiene il primo vero elemento della sequenza, così come il campo *prev* del nodo *trailer* farà riferimento al nodo che contiene l'ultimo elemento della sequenza.

Vantaggio derivante dall'uso delle sentinelle

Anche se si potrebbe implementare una lista doppiamente concatenata senza i nodi sentinella (come abbiamo fatto con la lista semplicemente concatenata del Paragrafo 3.2), la piccola quantità di memoria dedicata alle sentinelle semplifica notevolmente la logica delle varie operazioni. Innanzitutto, i nodi che fungono da sentinella non cambiano mai: sono soltanto i nodi che si trovano tra di loro a cambiare. Inoltre, con la loro presenza siamo in grado di trattare tutte le operazioni di inserimento allo stesso modo, perché il nuovo nodo viene sempre inserito tra una coppia di nodi esistenti. Analogamente, ogni elemento che deve essere rimosso sarà necessariamente memorizzato in un nodo che ha almeno un nodo adiacente su ciascun lato.

Per fare un confronto, torniamo a esaminare la nostra implementazione di `SinglyLinkedList`, nel Paragrafo 3.2. Il suo metodo `addLast` ha bisogno di un enunciato condizionale (righe 39–42 del Codice 3.15) per gestire il caso speciale di inserimento in una lista vuota. Infatti, in generale il nuovo nodo viene inserito dopo l'ultimo nodo, ma, inserendo in una lista vuota, l'ultimo nodo non esiste: in tal caso, è necessario assegnare a `head` il riferimento al nuovo nodo. L'uso di un nodo sentinella in quella implementazione eliminerebbe il caso speciale, perché prima di un nodo ci sarebbe sempre un nodo già esistente (al limite, il nodo sentinella iniziale).

Inserimento e rimozione in una lista doppiamente concatenata

Nella nostra lista doppiamente concatenata, ogni inserimento avviene tra una coppia di nodi esistenti, come evidenziato nella Figura 3.20. Ad esempio, se viene inserito un nuovo elemento all'inizio della sequenza, il nodo che lo contiene si posizionerà *fra* il nodo intestazione e il nodo che attualmente risulta essere il successivo del nodo intestazione (Figura 3.21).

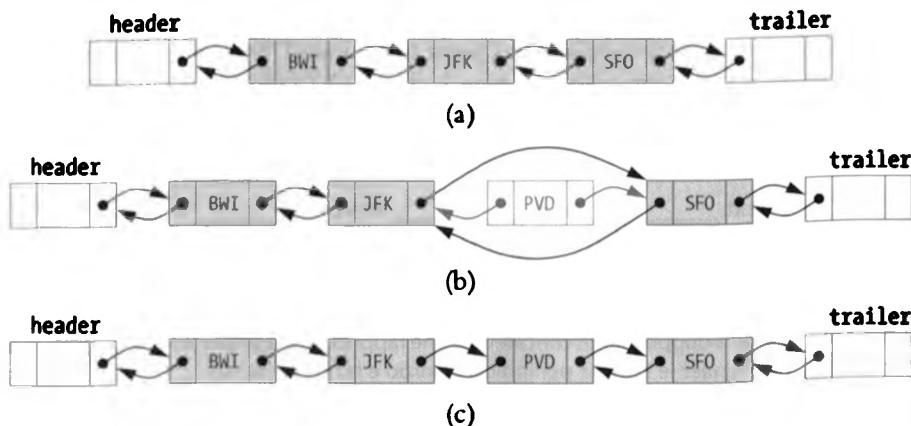


Figura 3.20: Aggiunta di un elemento a una lista doppiamente concatenata con nodi sentinella iniziale e finale: (a) prima dell'inserimento; (b) dopo la creazione del nuovo nodo; (c) dopo aver collegato il nuovo nodo ai nodi adiacenti.

L'eliminazione di un nodo, rappresentata nella Figura 3.22, procede in modo opposto all'inserimento. I due nodi adiacenti a quello da eliminare vengono collegati direttamente

più considerato come appartenente alla lista e la memoria che occupa potrà essere recuperata dal sistema, eliminando l'oggetto. Usando le sentinelle, si può utilizzare la medesima implementazione anche per eliminare il primo o l'ultimo elemento della sequenza, perché anche tali elementi sono memorizzati in un nodo che si trova tra due altri nodi.

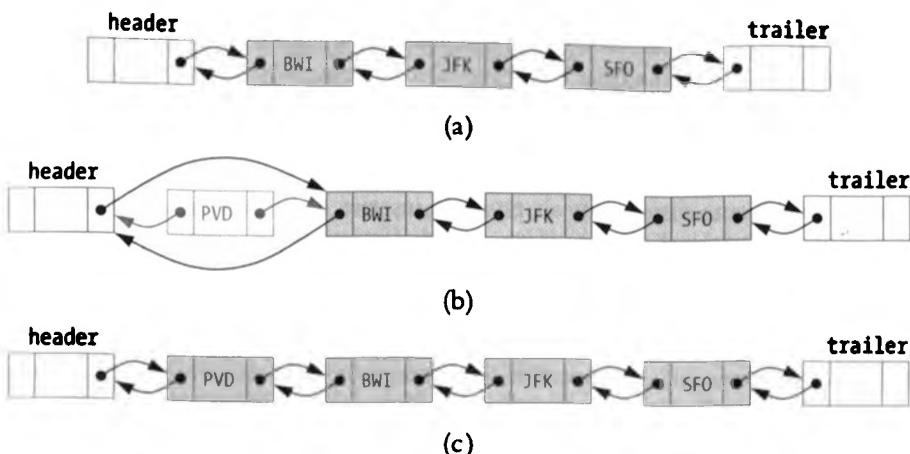


Figura 3.21: Aggiunta di un elemento all'inizio della sequenza rappresentata da una lista doppiamente concatenata con nodi sentinella iniziale e finale: (a) prima dell'inserimento; (b) dopo la creazione del nuovo nodo; (c) dopo aver collegato il nuovo nodo ai nodi adiacenti.

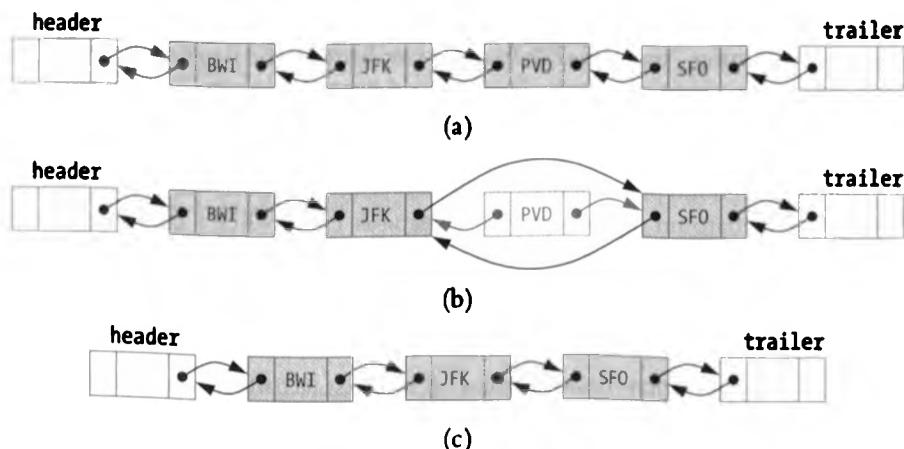


Figura 3.22: Eliminazione dell'elemento PVD da una lista doppiamente concatenata: (a) prima della rimozione; (b) dopo aver scollegato dalla lista il nodo che contiene PVD; (c) dopo che quel nodo è stato eliminato (dall'azione del *garbage collector* di Java).

3.4.1 Realizzare una lista doppiamente concatenata

In questo paragrafo presentiamo l'implementazione completa della classe `DoublyLinkedList`, che mette a disposizione i seguenti metodi pubblici:

- `size()`: Restituisce il numero di elementi presenti nella lista.
- `isEmpty()`: Restituisce `true` se e solo se la lista è vuota.
- `first()`: Restituisce il primo elemento della lista (senza eliminarlo).
- `last()`: Restituisce l'ultimo elemento della lista (senza eliminarlo).
- `addFirst(e)`: Aggiunge un nuovo elemento (`e`) all'inizio della lista.
- `addLast(e)`: Aggiunge un nuovo elemento (`e`) alla fine della lista.
- `removeFirst()`: Elimina dalla lista il suo primo elemento e lo restituisce.
- `removeLast()`: Elimina dalla lista il suo ultimo elemento e lo restituisce.

Se i metodi `first()`, `last()`, `removeFirst()` o `removeLast()` vengono invocati con una lista vuota, restituiscono semplicemente un riferimento `null`, lasciando la lista immutata.

Anche se abbiamo visto come sia possibile inserire o eliminare un elemento in qualsiasi posizione interna a una lista doppiamente concatenata, per farlo occorre conoscere un nodo (o più di uno) per identificare la posizione in cui deve avvenire l'operazione. In questo capitolo preferiamo preservare l'incapsulamento, usando una classe `Node` annidata e privata. Nel Capitolo 7 rivisiteremo l'uso delle liste doppiamente concatenate, offrendo un'interfaccia più avanzata che consentirà inserimenti e rimozioni in posizioni interne, pur mantenendo l'incapsulamento.

Le sezioni di Codice 3.17 e 3.18 presentano l'implementazione della classe `DoublyLinkedList`. Come abbiamo fatto per la classe `SinglyLinkedList`, usiamo la programmazione per tipi generici, in modo che una lista possa accettare elementi di qualsiasi tipo. La classe `Node` annidata nella lista doppiamente concatenata è simile a quella della lista semplicemente concatenata, ma ha un riferimento aggiuntivo, `prev`, che punta al nodo precedente.

La nostra decisione di usare i nodi `sentinella`, `header` e `trailer`, agisce sull'implementazione in vari modi. Le sentinelle vengono create e collegate nel momento in cui viene costruita una lista vuota (righe 25-29). Inoltre, ricordiamoci che il primo elemento di una lista non vuota viene memorizzato nel nodo che si trova subito *dopo* l'intestazione (e non nell'intestazione stessa), così come l'ultimo elemento è memorizzato nel nodo che *precede* la sentinella finale.

Le sentinelle rendono più semplice l'implementazione dei vari metodi di aggiornamento. Abbiamo progettato un metodo privato, `addBetween`, che gestisce il caso più generale di inserimento tra due nodi, per poi sfruttarlo in modo da rendere quasi banale l'implementazione dei metodi `addFirst` e `addLast`. Analogamente, abbiamo definito il metodo privato `remove`, usato per realizzare facilmente i metodi `removeFirst` e `removeLast`.

Codice 3.17: Implementazione della classe `DoublyLinkedList` (la parte rimanente della classe si trova nel Codice 3.18).

```

1  /** Un'implementazione basilare di lista doppiamente concatenata. */
2  public class DoublyLinkedList<E> {
3      //----- classe annidata Node -----
4      private static class Node<E> {
5          private E element;    // riferimento all'elemento memorizzato in questo nodo
6          private Node<E> prev; // riferimento al nodo precedente nella lista
7          private Node<E> next; // riferimento al nodo seguente nella lista
8          public Node(E e, Node<E> p, Node<E> n) {
9              element = e;
10             prev = p;
11             next = n;

```

```

12 }
13 public E getElement() { return element; }
14 public Node<E> getPrev() { return prev; }
15 public Node<E> getNext() { return next; }
16 public void setPrev(Node<E> p) { prev = p; }
17 public void setNext(Node<E> n) { next = n; }
18 } //----- fine della classe annidata Node -----
19
20 // variabili di esemplare di DoublyLinkedList
21 private Node<E> header;      // sentinella iniziale
22 private Node<E> trailer;     // sentinella finale
23 private int size = 0;         // numero di elementi della lista
24 /** Costruisce una nuova lista vuota. */
25 public DoublyLinkedList() {
26     header = new Node<E>(null, null, null);    // crea il nodo header
27     trailer = new Node<E>(null, header, null); // crea trailer preceduto da header
28     header.setNext(trailer);                  // così header è seguito da trailer
29 }
30 /** Restituisce il numero di elementi presenti nella lista. */
31 public int size() { return size; }
32 /** Restituisce true se e solo se la lista è vuota. */
33 public boolean isEmpty() { return size == 0; }
34 /** Restituisce il primo elemento della lista, senza eliminarlo. */
35 public E first() {
36     if (isEmpty()) return null;
37     return header.getNext().getElement(); // il primo elemento segue header
38 }
39 /** Restituisce l'ultimo elemento della lista, senza eliminarlo. */
40 public E last() {
41     if (isEmpty()) return null;
42     return trailer.getPrev().getElement(); // l'ultimo elemento precede trailer
43 }

```

Codice 3.18: Implementazione della classe DoublyLinkedList (prosegue dal Codice 3.17).

```

44 // metodi di aggiornamento pubblici
45 /** Aggiunge un elemento all'inizio della lista. */
46 public void addFirst(E e) {
47     addBetween(e, header, header.getNext()); // segue immediatamente header
48 }
49 /** Aggiunge un elemento alla fine della lista. */
50 public void addLast(E e) {
51     addBetween(e, trailer.getPrev(), trailer); // precede immediatamente trailer
52 }
53 /** Elimina e restituisce il primo elemento della lista. */
54 public E removeFirst() {
55     if (isEmpty()) return null;           // non c'è niente da eliminare
56     return remove(header.getNext()); // il primo è il successivo di header
57 }
58 /** Elimina e restituisce l'ultimo elemento della lista. */
59 public E removeLast() {
60     if (isEmpty()) return null;           // non c'è niente da eliminare
61     return remove(trailer.getPrev()); // l'ultimo è il predecessore di trailer
62 }
63
64 // metodi di aggiornamento privati
65 /** Aggiunge l'elemento e alla lista, tra i due nodi dati. */
66 private void addBetween(E e, Node<E> predecessor, Node<E> successor) {

```

```

67 // crea un nuovo nodo e lo collega alla lista
68 Node<E> newest = new Node<E>(e, predecessor, successor);
69 predecessor.setNext(newest);
70 successor.setPrev(newest);
71 size++;
72 }
73 /** Elimina dalla lista il nodo indicato e restituisce il suo elemento. */
74 private E remove(Node<E> node) {
75     Node<E> predecessor = node.getPrev();
76     Node<E> successor = node.getNext();
77     predecessor.setNext(successor);
78     successor.setPrev(predecessor);
79     size--;
80     return node.getElement();
81 }
82 } //----- fine della classe DoublyLinkedList -----

```

3.5 Verifica di equivalenza

Quando si lavora con i riferimenti, ci sono vari modi per interpretare il fatto che un'espressione sia uguale a un'altra. Per iniziare, se `a` e `b` sono variabili riferimento, l'espressione `a == b` verifica se `a` e `b` fanno riferimento allo stesso oggetto (o se `a` entrambe è stato assegnato il valore `null`).

Per molti tipi di dati (e, quindi, di riferimenti a essi) esiste una nozione di più alto livello di "equivalenza" tra due variabili, che può esistere anche se non fanno riferimento al medesimo esemplare della classe. Ad esempio, tipicamente diciamo che due esemplari di `String` sono equivalenti se rappresentano la stessa sequenza di caratteri.

Per dar forma a un concetto di equivalenza più ampio, tutti i tipi di oggetti dispongono di un metodo che si chiama `equals`. A meno che un programmatore non abbia veramente la necessità di verificare il concetto di identità più ristretto che abbiamo menzionato all'inizio del paragrafo, con `a == b`, si dovrebbe sempre usare la sintassi `a.equals(b)`. Formalmente, il metodo `equals` è definito nella classe `Object`, che svolge il ruolo di superclasse per tutti i tipi di riferimenti, ma quella implementazione, in effetti, restituisce il valore dell'espressione `a == b`: per definire un concetto di equivalenza più utile, è necessario conoscere la classe e la sua rappresentazione dei dati.

L'autore di ogni classe ha la responsabilità di fornire un'implementazione del metodo `equals`, che sovrascriva quella ereditata da `Object`, se esiste una definizione più significativa di equivalenza tra due esemplari della classe stessa. Ad esempio, nella libreria di Java la classe `String` ridefinisce `equals` in modo che verifichi l'equivalenza di due stringhe carattere per carattere.

Nel ridefinire (mediante sovrascrittura di `equals`) il concetto di uguaglianza o equivalenza occorre fare molta attenzione, perché la coerenza della libreria di Java si basa anche sulla definizione del metodo `equals`, che deve realizzare una *relazione di equivalenza* in senso matematico, soddisfacendo le seguenti proprietà:

Gestione di `null` Per qualunque variabile riferimento `x` non nulla, l'invocazione `x.equals(null)` deve restituire `false` (perché nulla, tranne `null`, è uguale a `null`).

- Riflessività** Per qualunque variabile riferimento `x` non nulla, l'invocazione `x.equals(x)` deve restituire `true` (perché un oggetto è uguale a se stesso).
- Simmetria** Per qualunque coppia di variabili riferimento `x` e `y` non nulle, le invocazioni `x.equals(y)` e `y.equals(x)` devono restituire lo stesso valore.
- Transitività** Per qualunque terna di variabili riferimento `x`, `y` e `z` non nulle, se entrambe le invocazioni `x.equals(y)` e `y.equals(z)` restituiscono `true`, allora anche l'invocazione `x.equals(z)` deve restituire `true`.

Anche se queste proprietà sembrano intuitive, per alcune strutture dati l'implementazione di `equals` può risultare complicata, specialmente in un contesto orientato agli oggetti che usi l'ereditarietà e la programmazione per tipi generici. Per la maggioranza delle strutture dati presentate in questo libro eviteremo di fornire una valida implementazione di `equals`, lasciandola come esercizio, ma in questo paragrafo ci occuperemo della verifica di equivalenza sia per gli array sia per le liste concatenate, presentando anche un esempio concreto di implementazione valida del metodo `equals` per la nostra classe `SinglyLinkedList`.

3.5.1 Verifica di equivalenza tra array

Come detto nel Paragrafo 1.3, in Java gli array sono tipi riferimento, ma non sono tecnicamente esemplari di una classe. La classe `java.util.Arrays` (presentata nel Paragrafo 3.1.3), però, mette a disposizione alcuni metodi statici che possono essere utili nell'elaborazione di array. Vediamo, quindi, un riassunto delle verifiche di equivalenza possibili per gli array, nell'ipotesi che `a` e `b` siano riferimenti a oggetti di tipo array:

- `a == b`: Verifica se `a` e `b` fanno riferimento allo stesso esemplare di array.
- `a.equals(b)`: È interessante notare come questa sintassi abbia lo stesso effetto di `a == b`: gli array non sono esemplari di una classe e, quindi, non dispongono di un metodo `equals` che sovrascriva `Object.equals`.
- `Arrays.equals(a,b)`: Questa invocazione è relativa a una nozione di equivalenza più intuitiva, restituendo `true` se e solo se gli array hanno la stessa lunghezza e tutte le coppie di elementi corrispondenti sono costituite da due elementi che sono "uguali" tra loro. Nello specifico, se gli elementi dell'array sono di un tipo di dato fondamentale, i loro valori vengono confrontati con il normale operatore `==`. Se, invece, gli elementi degli array sono riferimenti, l'equivalenza tra gli array viene valutata invocando, per ogni coppia di elementi, il metodo `a[k].equals(b[k])`.

Per la maggior parte delle applicazioni, il comportamento di `Arrays.equals` rappresenta adeguatamente il concetto di equivalenza, ma, quando si usano array a più dimensioni, c'è qualche problema in più. Il fatto che gli array bidimensionali, in Java, siano in realtà array monodimensionali annidati all'interno di un normale array monodimensionale ci fa riflettere sugli *oggetti composti*, che sono oggetti, come gli array bidimensionali, che sono

costituiti da altri oggetti. In particolare, dobbiamo decidere dove inizia e dove termina un oggetto composto.

Infatti, se confrontiamo due array bidimensionali, `a` e `b`, che hanno gli stessi elementi, probabilmente vogliamo poter ritenere `a` equivalente a `b`, ma gli array monodimensionali che costituiscono le righe di `a` e `b` (come, ad esempio, `a[0]` e `b[0]`) sono memorizzati in zone diverse della memoria, anche nel caso in cui abbiano lo stesso contenuto, quindi l'invocazione del metodo `java.util.Arrays.equals(a, b)` restituirà `false`, perché andrà a invocare `a[k].equals(b[k])`, metodo che, a sua volta, invocherà la definizione di `equals` che si trova nella classe `Object`.

Per consentire di estendere anche agli array multidimensionali il concetto di equivalenza che ci appare più naturale, la classe `java.util.Arrays` definisce un ulteriore metodo:

`Arrays.deepEquals(a,b):` Ha lo stesso comportamento di `Arrays.equals(a, b)`, tranne quando gli elementi di `a` e `b` sono a loro volta array, nel qual caso, per ogni coppia di array corrispondenti, invece di invocare `a[k].equals(b[k])`, invoca `Arrays.deepEquals(a[k], b[k])`.

3.5.2 Verifica di equivalenza tra liste concatenate

In questo paragrafo svilupperemo un'implementazione del metodo `equals` per la classe `SinglyLinkedList` vista nel Paragrafo 3.2.1. Usando una definizione molto simile a quella utilizzata per gli array dal metodo `java.util.Arrays.equals`, consideriamo due liste equivalenti se hanno la stessa lunghezza e i loro contenuti sono equivalenti elemento per elemento. Una tale equivalenza può essere valutata scandendo simultaneamente le due liste e verificando se `x.equals(y)` restituisce `true` per ogni coppia di elementi `x` e `y` corrispondenti.

Codice 3.19: Implementazione del metodo `SinglyLinkedList.equals`.

```

1  public boolean equals(Object o) {
2      if (o == null) return false;
3      if (getClass() != o.getClass()) return false;
4      SinglyLinkedList other = (SinglyLinkedList) o; // tipo non parametrico
5      if (size != other.size) return false;
6      Node walkA = head; // scandisce la prima lista
7      Node walkB = other.head; // scandisce la seconda lista
8      while (walkA != null) {
9          if (!walkA.getElement().equals(walkB.getElement())) return false; // differenza
10         walkA = walkA.getNext();
11         walkB = walkB.getNext();
12     }
13     return true; // arrivati qui, tutti i confronti sono andati bene
14 }
```

L'implementazione del metodo `SinglyLinkedList.equals` è riportata nel Codice 3.19. Nonostante ci stiamo occupando nel dettaglio del confronto di due liste semplicemente concatenate, il metodo `equals`, per sovrascrivere quello ereditato da `Object`, deve ricevere come parametro un riferimento di tipo `Object`. Seguiamo un approccio conservativo, richiedendo che due oggetti siano esemplari della stessa classe perché possano essere considerati equivalenti (ad esempio, non consideriamo equivalenti una lista semplicemente concatenata

è una lista doppiamente concatenata che memorizzino la stessa sequenza di elementi). Dopo aver verificato, alla riga 2, che il parametro o non sia nullo, la riga 3 usa il metodo `equals()`, disponibile per qualsiasi oggetto, per verificare se i due oggetti in esame sono esemplari della stessa classe.

Arrivati alla riga 4, siamo sicuri che il parametro ricevuto è un esemplare della classe `SinglyLinkedList` (o di una sua sottoclasse), per cui possiamo eseguire senza problemi il cast per ottenere un riferimento di tipo `SinglyLinkedList`, con il quale si potrà poi accedere alle variabili di esemplare `size` e `head`.

Occorre qui sottolineare un dettaglio relativo alla gestione, in Java, dell'infrastruttura di programmazione per tipi generici. Nonostante la nostra classe `SinglyLinkedList` abbia dichiarato il parametro formale di tipo, `<E>`, non è possibile verificare, al momento dell'esecuzione, se l'altra lista con cui effettuiamo il confronto ha lo stesso parametro effettivo (chi fosse interessato, può cercare informazioni in merito al fenomeno di *erasure* in Java). Di conseguenza, dobbiamo ricadere nell'approccio classico, usando, alla riga 4, il tipo `SinglyLinkedList` non parametrico, e, alle righe 6 e 7, la dichiarazione di `Node` non parametrico. Se le due liste hanno tipi incompatibili, la cosa verrà rilevata durante l'invocazione del metodo `equals` su elementi corrispondenti.

3.6 Clonare strutture dati

L'eleganza della programmazione orientata agli oggetti risiede principalmente nell'astrazione, che consente di trattare una struttura dati come oggetto atomico, anche se la sua struttura incapsulata può basarsi su una ben più complessa combinazione di molti oggetti. In questo paragrafo vedremo come si possa fare una copia di una tale struttura.

In un ambiente di programmazione, è lecito aspettarsi che la copia di un oggetto disponga di un proprio stato e che, una volta eseguita la copia, questa sia indipendente dall'originale (in modo che, ad esempio, se la copia viene modificata, l'originale non si modifica). Quando, però, gli oggetti contengono campi che sono variabili riferimento che puntano ad altri oggetti, non sempre è chiaro se una copia avrà corrispondentemente campi che fanno riferimento agli stessi oggetti oppure a una loro copia.

Ad esempio, se un'ipotetica classe `AddressBook` consente di creare esemplari che rappresentano una rubrica elettronica, contenente informazioni (come il numero di telefono e l'indirizzo di posta elettronica) degli amici e conoscenti di una persona, come possiamo immaginare di fare una copia di tale oggetto? Aggiungendo un nuovo contatto a uno dei due oggetti (copia o originale), questo deve comparire anche nell'altro? Se in una rubrica cambiamo il numero di telefono di una persona, ci aspettiamo che tale modifica avvenga in modo sincrono anche nell'altra?

Domande come queste non hanno risposte che vanno bene in tutte le situazioni: piuttosto, ogni classe Java deve avere la responsabilità di definire se i propri esemplari possono essere copiati e, in caso affermativo, descrivere con precisione come si costruisca la copia. La superclasse universale `Object` definisce un metodo, `clone`, che può essere utilizzato per generare quella che viene chiamata *copia superficiale* (*shallow copy*) di un oggetto. Tale metodo usa la normale semantica dell'assegnazione per dare un valore a ciascun campo del nuovo oggetto, in modo che sia uguale al campo corrispondente dell'oggetto che viene copiato. Si

parla di “copia superficiale” perché, se un campo è di tipo riferimento, la sua inizializzazione `duplicate.field = original.field` fa in modo che il campo (`field`) del nuovo oggetto (`duplicate`) si riferisca allo stesso esemplare di oggetto a cui fa riferimento il campo `field` dell’oggetto originale (`original`).

Non sempre la copia superficiale è appropriata per una classe, quindi Java disabilita intenzionalmente l’uso del metodo `clone()` dichiarandolo `protected` e lanciando l’eccezione `CloneNotSupportedException` quando viene invocato. L’autore di una classe deve dichiarare esplicitamente di voler consentire la clonazione dei suoi esemplari, indicando formalmente che la classe implementa l’interfaccia `Cloneable` (che ha l’effetto di evitare che l’invocazione del metodo `protected clone()` ereditato lanci l’eccezione, perché così è definito quel metodo) e definendo una versione pubblica del metodo `clone()`. Tale metodo pubblico, se l’autore lo ritiene opportuno, può semplicemente invocare il metodo ereditato, che farà un’assegnazione campo per campo, generando così una copia superficiale. Altrimenti, come avviene in effetti per molte classi, il progettista può scegliere di implementare una versione di clonazione meno superficiale (o, come si dice, una *copia profonda, deep copy*), nella quale alcuni (o tutti) gli oggetti a cui si riferiscono i campi vengono a loro volta clonati.

Nella maggior parte delle strutture dati presentate in questo libro abbiamo omesso l’implementazione di un metodo `clone` valido (lasciadola come esercizio), ma in questo paragrafo vedremo soluzioni per clonare array e liste concatenate, tra le quali presenteremo un’implementazione completa del metodo `clone` per la classe `SinglyLinkedList`.

3.6.1 Clonare array

Anche se gli array consentono l’utilizzo di alcune strutture sintattiche speciali, come `a[k]` e `a.length`, è importante ricordare che sono oggetti e che le “variabili array” sono in ogni caso variabili riferimento, con alcune conseguenze importanti. Come primo esempio, analizziamo il codice seguente

```
int[] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[] backup;
backup = data; // attenzione: non è una copia
```

L’assegnazione del valore di `data` alla variabile `backup` non crea un nuovo array, crea semplicemente un modo alternativo per accedere allo stesso array o, come si dice, un *alias*, la situazione rappresentata nella Figura 3.23.

Se, invece, vogliamo creare una copia dell’array `data` e assegnare alla variabile `backup` il riferimento al nuovo array, dobbiamo scrivere:

```
backup = data.clone();
```

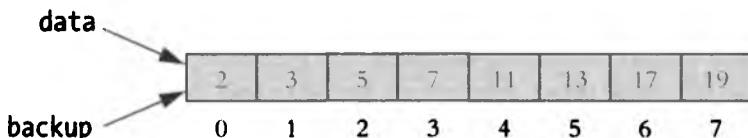


Figura 3.23: Risultato prodotto dall’assegnazione `backup = data` tra due variabili di tipo riferimento ad array di `int`.

Il metodo `clone`, quando viene eseguito su un array, inizializza ciascuna cella del nuovo array al valore memorizzato nella cella corrispondente dell'array originale, dando così luogo a un array indipendente, come mostrato nella Figura 3.24.



Figura 3.24: Risultato prodotto dall'assegnazione `backup = data.clone()` tra due variabili di tipo riferimento ad array di `int`.

Se, dopo la clonazione, facciamo un'assegnazione come `data[4] = 23`, l'array `backup` non viene modificato.

Quando si copia un array che memorizza riferimenti, invece che valori di un tipo fondamentale, le considerazioni da fare si complicano: il metodo `clone()` genera una *copia superficiale* dell'array, cioè genera un nuovo array le cui celle fanno riferimento agli stessi oggetti a cui fanno riferimento le celle dell'array originale.

Ad esempio, se la variabile `contacts` si riferisce a un array di esemplari di un'ipotetica classe `Person`, l'assegnazione `guests = contacts.clone()` genera una copia superficiale di `contacts`, come si può vedere nella Figura 3.25.

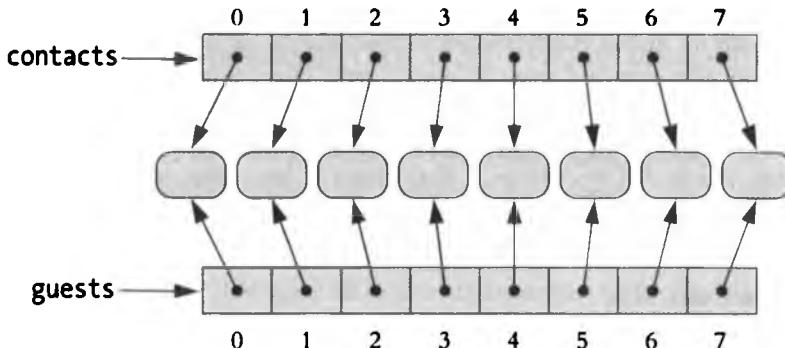


Figura 3.25: La copia superficiale di un array di oggetti, prodotta dall'assegnazione `guests = contacts.clone()`.

Per creare una *copia profonda* dell'array `contacts` si possono clonare i suoi singoli elementi, con un ciclo come il seguente, ma tale operazione può avere successo soltanto se la classe `Person` implementa l'interfaccia `Cloneable`:

```
Person[] guests = new Person[contacts.length];
for (int k=0; k < contacts.length; k++)
    guests[k] = (Person) contacts[k].clone(); // restituisce un riferimento Object
```

Dal momento che un array bidimensionale è, in realtà, un array monodimensionale che memorizza, come propri elementi, altri array monodimensionali, è di nuovo in essere la distinzione tra copia superficiale e copia profonda. Sfortunatamente la classe `java.util.Arrays` non contiene un metodo “`deepClone`”, ma lo possiamo implementare clonando le singole righe di un array, come si può vedere nel Codice 3.20 nel caso di un array bidimensionale di numeri interi.

Codice 3.20: Un metodo che crea una copia profonda di un array bidimensionale di numeri interi.

```

1 public static int[][] deepClone(int[][] original) {
2     int[][] backup = new int[original.length][];
3     for (int k=0; k < original.length; k++)
4         backup[k] = original[k].clone();           // copia la riga k
5     return backup;
6 }
```

3.6.2 Clonare liste concatenate

In questo paragrafo aggiungeremo alla classe `SinglyLinkedList` del Paragrafo 3.2.1 la possibilità di clonare propri esemplari. In Java, il primo passo da compiere per rendere clonabile una classe è l'aggiunta della dichiarazione di implementazione dell'interfaccia `Cloneable`, quindi modifichiamo la prima riga della definizione della classe in modo che diventi questa:

```
public class SinglyLinkedList<E> implements Cloneable {
```

Poi, quello che manca è l'implementazione, nella classe, di una versione pubblica del metodo `clone()`, che presentiamo nel Codice 3.21. Per convenzione, tale metodo deve iniziare (riga 3) creando un nuovo esemplare della classe usando un'invocazione di `super.clone()`, che, nel nostro caso, invoca il metodo `clone()` della classe `Object`. Dato che tale versione ereditata restituisce un riferimento di tipo `Object`, effettuiamo un cast con restrizione per trasformarlo in un riferimento di tipo `SinglyLinkedList<E>`.

A questo punto dell'esecuzione, è stata creata la lista `other` come copia superficiale dell'originale. Dato che la nostra classe che rappresenta le liste ha due campi, `size` e `head`, sono state eseguite le seguenti assegnazioni:

```
other.size = this.size;
other.head = this.head;
```

Mentre l'assegnazione che coinvolge la variabile `size` è corretta, non possiamo lasciare che la nuova lista condivida con l'originale lo stesso valore di `head` (a meno che non sia `null`). Perché una lista non vuota disponga di uno stato indipendente, deve avere una catena di nodi completamente nuova, con ciascun nodo che memorizza al proprio interno un riferimento al corrispondente elemento presente nella lista originale. Quindi, per prima cosa alla riga 5 creiamo un nuovo nodo iniziale della catena, poi eseguiamo una scansione della parte restante della lista originale (righe 8-13), creando e collegando contemporaneamente i nuovi nodi per la nuova lista.

Codice 3.21: Implementazione del metodo `SinglyLinkedList.clone()`.

```

1  public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
2      // per creare la copia iniziale si usa sempre il metodo ereditato
3      SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // cast valido
4      if (size > 0) {                                // ci serve una catena di nodi indipendente
5          other.head = new Node<E>(head.getElement(), null);
6          Node<E> walk = head.getNext(); // scandisce la lista originale
7          Node<E> otherTail = other.head; // ricorda il nodo creato più recentemente
8          while (walk != null) {           // crea un nuovo nodo per lo stesso elemento
9              Node<E> newest = new Node<E>(walk.getElement(), null);
10             otherTail.setNext(newest);    // collega il nodo precedente a questo
11             otherTail = newest;
12             walk = walk.getNext();
13         }
14     }
15     return other;
16 }
```

3.7 Esercizi

Riepilogo e approfondimento

- R-3.1** Individuare i cinque numeri pseudocasuali successivi a quelli generati dal processo descritto nel Paragrafo 3.1.3, con $a = 12$, $b = 5$, $n = 100$ e 92 come seme per `cur`.
- R-3.2** Scrivere un metodo Java che rimuova ripetutamente da un array un dato scelto a caso, finché l'array non rimane vuoto.
- R-3.3** Spiegare le modifiche che sarebbe necessario apportare al programma del Codice 3.8 in modo che possa eseguire la cifratura di Cesare per messaggi scritti in un linguaggio alfabetico diverso dall'inglese, come il greco, il russo o l'ebraico.
- R-3.4** La classe `TicTacToe` presentata nel Codice 3.9 e 3.10 ha un difetto, perché consente a un giocatore di posizionare una pedina anche dopo che la partita è stata vinta da qualcuno. Modificare la classe in modo che, in una situazione come quella descritta, il metodo `putMark` lanci un'eccezione di tipo `IllegalStateException`.
- R-3.5** Il metodo `removeFirst` della classe `SinglyLinkedList` prevede, come caso speciale, che il campo `tail` venga riportato al valore `null` quando dalla lista viene eliminato l'ultimo nodo (righe 51 e 52 del Codice 3.15). Che conseguenze si avrebbero se eliminassimo quelle due righe dal codice? Spiegare perché la classe funzionerebbe o non funzionerebbe dopo tale modifica.
- R-3.6** Descrivere un algoritmo che cerca il penultimo nodo di una lista semplicemente concatenata il cui ultimo nodo è identificato soltanto dal fatto di avere un riferimento `null` nel campo `next`.
- R-3.7** Analizzare l'implementazione del metodo `CircularlyLinkedList.addFirst`, nel Codice 3.16. Il corpo della clausola `else` alle righe 39 e 40 si basa su una variabile dichiarata localmente, `newest`. Riprogettare tale clausola in modo che si eviti l'utilizzo di variabili locali.
- R-3.8** Descrivere un metodo che cerca il nodo centrale di una lista doppiamente concatenata dotata di sentinelle iniziale e finale, usando la strategia di "link hopping", senza basarsi in modo esplicito sulla dimensione della lista. Nel caso in cui il numero

di nodi sia pari, individuare come "nodo centrale" quello più a sinistra tra i due nodi centrali. Qual è il tempo di esecuzione di questo metodo?

- R-3.9 Fornire un'implementazione del metodo `size()` della classe `SinglyLinkedList`, nell'ipotesi che la dimensione non sia memorizzata in una variabile di esemplare.
- R-3.10 Fornire un'implementazione del metodo `size()` della classe `CircularlyLinkedList`, nell'ipotesi che la dimensione non sia memorizzata in una variabile di esemplare.
- R-3.11 Fornire un'implementazione del metodo `size()` della classe `DoublyLinkedList`, nell'ipotesi che la dimensione non sia memorizzata in una variabile di esemplare.
- R-3.12 Implementare il metodo `rotate()` nella classe `SinglyLinkedList`, in modo che abbia lo stesso comportamento dell'invocazione `addLast(removeFirst())`, senza, però, che venga creato alcun nuovo nodo.
- R-3.13 Che differenza c'è, in Java, tra una verifica di equivalenza superficiale e una profonda, nel caso di due array, *A* e *B*, monodimensionali di tipo `int`? E se i due array sono bidimensionali, sempre di tipo `int`?
- R-3.14 Fornire tre diversi esempi di un unico enunciato che, in Java, assegna alla variabile `backup` il riferimento a un nuovo array che contiene copie di tutti i valori di tipo `int` di un altro array preesistente, `original`.
- R-3.15 Implementare il metodo `equals` per la classe `CircularlyLinkedList`, nell'ipotesi che due liste siano uguali se contengono la stessa sequenza di elementi, con elementi corrispondenti che si trovano all'inizio della lista.
- R-3.16 Implementare il metodo `equals` per la classe `DoublyLinkedList`.

Creatività

- C-3.17 Sia *A* un array di dimensione $n \geq 2$ contenente i numeri interi da 1 a $n - 1$, estremi compresi, uno dei quali è ripetuto. Descrivere un algoritmo che trova il numero ripetuto in *A*.
- C-3.18 Sia *B* un array di dimensione $n \geq 6$ contenente i numeri interi da 1 a $n - 5$, estremi compresi, cinque dei quali sono ripetuti. Descrivere un algoritmo che trova i cinque numeri ripetuti in *B*.
- C-3.19 Progettare il codice Java dei metodi `add(e)` e `remove(e)` della classe `Scoreboard`, vista nel Codice 3.3. e 3.4, nell'ipotesi che, questa volta, i punteggi non vengano conservati in ordine. Ipotizzare che, di nuovo, si debbano memorizzare *n* punteggi nelle celle di un array aventi indici da 0 a $n - 1$. Non si devono usare cicli, in modo che il numero di passi da eseguire non dipenda da *n*.
- C-3.20 In relazione al generatore pseudocasuale visto nel Paragrafo 3.1.3, fornire esempi di valori di *a* e *b* tali che, per $n = 1000$, la sequenza generata non sembri affatto casuale.
- C-3.21 Ipotizzare che l'array *A* contenga 100 numeri interi generati usando il metodo `r.nextInt(10)`, dove *r* è un oggetto di tipo `java.util.Random`. Sia *x* il prodotto di tutti i numeri presenti in *A*: esiste un unico numero a cui *x* sarà uguale con probabilità non inferiore a 0.99. Qual è quel numero e qual è una formula che descrive la probabilità che *x* sia uguale a quel numero?
- C-3.22 Usando il metodo `nextInt(n)` della classe `java.util.Random`, che restituisce un numero casuale compreso tra 0 e $n - 1$, estremi inclusi, scrivere un metodo, `shuffle(A)`, che mescoli gli elementi dell'array *A* in modo che qualunque possibile loro disposizione sia ugualmente probabile come risultato.

- C-3.23 Si immagini di dover progettare un gioco che, in modalità *multiplayer*, consenta la partecipazione simultanea di $n \geq 1000$ giocatori, numerati da 1 a n , interagenti all'interno di una foresta incantata. Il vincitore del gioco è il primo giocatore che riesce a incontrare nella foresta tutti gli altri giocatori almeno una volta (sono ammesse situazioni di parità). Nell'ipotesi che sia disponibile il metodo `meet(i, j)`, che viene invocato ogni volta che il giocatore i incontra il giocatore j (con $i \neq j$), descrivere una strategia per tenere traccia delle coppie di giocatori che si sono incontrati, per poter decretare il vincitore.
- C-3.24 Scrivere un metodo Java che riceve come parametri due array tridimensionali di numeri interi e li somma elemento per elemento.
- C-3.25 Descrivere un algoritmo che concatensi due liste semplicemente concatenate, L e M , generando un'unica lista, L' , che contenga ordinatamente tutti i nodi di L seguiti da tutti i nodi di M .
- C-3.26 Descrivere un algoritmo che concatensi due liste doppiamente concatenate, L e M , dotate di nodi sentinella iniziale e finale, generando un'unica lista, L' , che contenga ordinatamente tutti i nodi di L seguiti da tutti i nodi di M .
- C-3.27 Descrivere in dettaglio come scambiare tra loro due nodi x e y (e non soltanto il loro contenuto) all'interno di una lista semplicemente concatenata L , essendo noti soltanto i riferimenti a x e y . Ripetere l'esercizio nel caso in cui L sia una lista doppiamente concatenata. Quale algoritmo richiede più tempo per essere eseguito?
- C-3.28 Descrivere in dettaglio un algoritmo che inverta una lista semplicemente concatenata L usando soltanto una quantità di spazio aggiuntivo costante.
- C-3.29 Date due liste concatenate circolari, L e M , descrivere un algoritmo che sia in grado di affermare se L e M memorizzano la stessa sequenza di elementi (eventualmente con posizioni-iniziali diverse).
- C-3.30 Data una lista concatenata circolare L contenente un numero pari di nodi, descrivere come la si possa suddividere in due liste concatenate circolari di dimensione pari alla metà della dimensione di L .
- C-3.31 La nostra implementazione di lista doppiamente concatenata si basa sulla presenza di due nodi sentinella, `header` e `trailer`, ma sarebbe sufficiente un unico nodo sentinella per "sorvegliare" entrambe le estremità della lista. Riprogettare la classe `DoublyLinkedList` in modo che usi un solo nodo sentinella.
- C-3.32 Realizzare una versione circolare di lista doppiamente concatenata, senza alcuna sentinella, che preveda il medesimo comportamento pubblico di quella originale, con l'aggiunta di due ulteriori metodi di aggiornamento, `rotate()` (come in `CircularlyLinkedList`) e `rotateBackward()` (che fa ruotare la lista in senso opposto).
- C-3.33 Risolvere il problema precedente usando l'ereditarietà, in modo che la classe `DoublyLinkedList` erediti dalla classe `CircularlyLinkedList` esistente e che la classe annidata `DoublyLinkedList.Node` erediti da `CircularlyLinkedList.Node`.
- C-3.34 Implementare il metodo `clone()` nella classe `CircularlyLinkedList`.
- C-3.35 Implementare il metodo `clone()` nella classe `DoublyLinkedList`.

Progettazione

- P-3.36 Scrivere un classe Java per rappresentare matrici, con un programma che sommi e moltipichi tra loro matrici bidimensionali di numeri interi.
- P-3.37 Scrivere una classe che gestisca i dieci punteggi migliori di un gioco elettronico, implementando i metodi `add` e `remove` visti nel Paragrafo 3.1.1, usando, però, una lista semplicemente concatenata invece di un array.
- P-3.38 Rifare il progetto precedente usando una lista doppiamente concatenata. In questo caso, inoltre, l'implementazione di `remove(i)` deve compiere il minimo numero di "salti" da un nodo all'altro per arrivare al punteggio di indice i .
- P-3.39 Scrivere un programma che sia in grado di eseguire la cifratura di Cesare per messaggi in lingua inglese che contengano caratteri maiuscoli e minuscoli.
- P-3.40 Implementare una classe, `SubstitutionCipher`, dotata di un costruttore che riceve come parametro una stringa di 26 lettere maiuscole in ordine arbitrario e la usa come codificatore in un cifrario (cioè la lettera A viene trasformata nella prima lettera del parametro, B viene trasformata nella seconda, e così via). La stringa da usare per decifrare va ricavata da quella usata per la cifratura.
- P-3.41 Riprogettare la classe `CaesarCipher` come sottoclasse della classe `SubstitutionCipher` realizzata come soluzione dell'esercizio precedente.
- P-3.42 Progettare la classe `RandomCipher` come sottoclasse della classe `SubstitutionCipher` realizzata come soluzione dell'Esercizio P-3.40, in modo che ciascun suo esemplare si basi, per la cifratura, su una permutazione casuale di lettere.
- P-3.43 Nel tradizionale gioco per bambini noto come "Duck, Duck, Goose" (*papera, papera, oca*), un gruppo di bambini si siede in cerchio. Uno di loro viene scelto a caso per essere di turno e cammina all'esterno del cerchio, toccando con la mano la testa degli altri bambini, uno alla volta seguendo il cerchio e dicendo "Duck" a ognuno, fino a quando non decide di dire "Goose" mentre tocca la testa di uno dei bambini seduti. A quel punto i due protagonisti (il bambino che era di turno e quello che è stato nominato "Goose") corrono attorno al cerchio, cercando di tornare per primi al posto da cui il "Goose" si è alzato e, infine, sedersi. Chi perde la gara rimane in piedi e diventa il successivo giocatore di turno. Il gioco continua in questo modo finché i bambini non si stancano o un adulto li chiama per la merenda. Scrivere un programma che simuli il gioco "Duck, Duck, Goose".

Note

Le strutture dati fondamentali, array e liste concatenate, discusse in questo capitolo appartengono ormai alla tradizione dell'informatica. Sono comparse per la prima volta nella letteratura scientifica nell'influenzante libro di Knuth, *Fundamental Algorithms* [60].

4

Analisi di algoritmi

Secondo la leggenda, al famoso matematico Archimede venne chiesto di determinare se una corona d'oro fatta costruire dal re fosse veramente d'oro puro, senza contenere argento, come sosteneva invece un informatore. Archimede scoprì come effettuare tale analisi mentre entrava nella vasca da bagno: notò che l'acqua fuoriusciva dalla vasca in proporzione alla parte del suo corpo che vi entrava. Ben consapevole delle implicazioni che questa scoperta potesse avere, uscì immediatamente dall'acqua e corse nudo per la città gridando "Eureka, eureka!" (in greco antico, "ho trovato"), perché aveva scoperto uno strumento di analisi (lo spostamento dell'acqua) che, insieme con una semplice scala graduata, poteva determinare se la nuova corona del re fosse d'oro oppure no. Infatti, Archimede poté immergersi in una vasca d'acqua prima la corona, poi una quantità d'oro avente lo stesso peso della corona, verificando se la quantità d'acqua fuoriuscita era la stessa. Questa scoperta scientifica portò, però, sfortuna all'orafo, perché, quando Archimede fece la sua analisi, la corona spostò più acqua della quantità d'oro puro avente lo stesso peso, segnalando che, in effetti, la corona non era d'oro puro.

In questo libro siamo interessati alla progettazione di "buone" strutture dati e "buoni" algoritmi. Detto semplicemente, una *struttura dati* (*data structure*) o, più precisamente, "struttura per memorizzare dati", è un modo sistematico per organizzare dati e accedervi, mentre un *algoritmo* è una procedura, descritta in più fasi o passi, che consente di risolvere un determinato problema in una quantità di tempo finita. Questi concetti sono centrali nell'informatica, ma, per essere in grado di qualificare strutture dati e algoritmi come "buoni", dobbiamo individuare metodi esatti per analizzarli.

Il principale strumento di analisi che useremo in questo libro riguarda la caratterizzazione dei tempi di esecuzione degli algoritmi e delle operazioni compiute sulle strutture dati, tenendo anche conto dello spazio di memoria utilizzato. Il tempo di esecuzione è una naturale misura di "bontà", perché il tempo è una risorsa preziosa: i programmi che

risolvono problemi mediante un computer devono terminare il proprio compito il più rapidamente possibile. In generale, il tempo di esecuzione di un algoritmo o di un'operazione su una struttura dati aumenta all'aumentare della dimensione dei dati in ingresso, anche se può essere diverso per dati in ingresso diversi, pur avendo la stessa dimensione. Il tempo di esecuzione è anche, ovviamente, funzione dell'ambiente usato per l'implementazione e l'esecuzione, tanto hardware (il processore, la frequenza di clock, la memoria, il disco, ecc.) quanto software (il sistema operativo, il linguaggio di programmazione, ecc.). A parità di tutti gli altri fattori, il tempo di esecuzione di uno stesso algoritmo operante sugli stessi dati sarà minore se il computer ha, ad esempio, un processore più veloce o se l'implementazione ha generato un programma compilato in linguaggio macchina nativo piuttosto che uno che necessiti di interpretazione per l'esecuzione su una macchina virtuale. Iniziamo questo capitolo presentando strumenti per l'esecuzione di analisi sperimentali, anche se l'uso di esperimenti come metodo principale di valutazione dell'efficienza di algoritmi, come vedremo, è piuttosto limitativo.

Concentrandoci sul tempo di esecuzione come misura principale della bontà di un algoritmo, ci sarà necessario poter utilizzare alcuni strumenti matematici. Nonostante le possibili variazioni che derivano da molteplici fattori, alcuni dei quali già citati, vorremmo porre l'attenzione sulla relazione esistente tra il tempo d'esecuzione di un algoritmo e la dimensione dei dati in ingresso: cercheremo di esprimere il tempo d'esecuzione come funzione della dimensione dei dati in ingresso. Ma in che modo si può misurare adeguatamente il tempo d'esecuzione? In questo capitolo ci "rimboccheremo le maniche" e svilupperemo un modello matematico per analizzare gli algoritmi.

4.1 Analisi sperimentali

Per studiare l'efficienza di un algoritmo lo si può implementare e, poi, si possono effettuare esperimenti, eseguendo il programma con varie configurazioni di dati in ingresso e registrando il tempo impiegato per ciascuna esecuzione. In Java, per raccogliere queste informazioni si può usare un meccanismo piuttosto semplice, che prevede l'utilizzo del metodo `currentTimeMillis` della classe `System`. Tale metodo restituisce il numero di millisecondi che sono trascorsi da un istante di tempo di riferimento, che, in informatica, prende il nome di "the epoch" ed è l'ora zero del primo gennaio del 1970, UTC. In effetti, non siamo particolarmente interessati al tempo trascorso da quel giorno: il punto chiave sta nel memorizzare, all'interno del programma, tale valore immediatamente prima dell'esecuzione dell'algoritmo, per poi valutarlo di nuovo subito dopo, misurando così il tempo *trascorso* durante l'esecuzione dell'algoritmo, come differenza tra quei due valori. Il Codice 4.1 riporta un esempio tipico di codice che svolge questa funzione.

Codice 4.1: Tipico esempio di codice che valuta il tempo di esecuzione di un algoritmo.

```

1 long startTime = System.currentTimeMillis(); // registra l'istante iniziale
2 /* esecuzione dell'algoritmo */
3 long endTime = System.currentTimeMillis(); // registra l'istante finale
4 long elapsed = endTime - startTime; // calcola il tempo trascorso

```

Misurando in questo modo il tempo trascorso si ottiene una valutazione abbastanza ragionevole dell'efficienza di un algoritmo; nel caso di operazioni estremamente veloci, Java consente l'utilizzo di un metodo, `nanoTime`, che misura il tempo in nanosecondi anziché in millisecondi.

Dato che siamo interessati alla relazione generale esistente tra il tempo d'esecuzione e la dimensione dei dati da elaborare, dovremmo eseguire esperimenti indipendenti con molti diversi dati in ingresso, di diverse dimensioni, per poi, ad esempio, visualizzare i risultati di ciascuna esecuzione dell'algoritmo sotto forma di grafico cartesiano, con la coordinata x che rappresenta la dimensione, n , dei dati in ingresso e la coordinata y che è uguale al tempo di esecuzione misurato, t . Un grafico di questo tipo consente spesso di intuire la relazione esistente tra la dimensione del problema e il tempo di esecuzione dell'algoritmo che lo risolve. Tutto questo può essere completato da un'analisi statistica che cerca di individuare la miglior funzione di n che approssima i dati sperimentali. Perché questa metodologia di analisi sia significativa, è necessario che vengano scelti con cura gli insiemi di dati da fornire in ingresso all'algoritmo, in numero sufficiente da rendere significativi i risultati statistici relativi al tempo d'esecuzione.

Tuttavia, i tempi d'esecuzione misurati tanto dal metodo `currentTimeMillis` quanto dal metodo `nanoTime` saranno fortemente variabili da macchina a macchina, e anche da un tentativo all'altro, anche se eseguiti sulla stessa macchina, per effetto dei molti processi che condividono la *CPU* (*central processing unit*, unità centrale di elaborazione) e la memoria di sistema di un computer. Quindi, il tempo trascorso durante l'esecuzione di un algoritmo dipenderà, in generale, da quali altri processi sono in esecuzione sul computer durante gli esperimenti. Anche se, per quanto appena detto, il tempo di esecuzione non può essere ritenuto affidabile nel suo valore misurato, gli esperimenti sono comunque utili nel momento in cui vengano utilizzati per confrontare l'efficienza di due o più algoritmi che risolvano lo stesso problema, almeno fintantoché vengono eseguiti in circostanze simili.

Come esempio concreto di analisi sperimentale, consideriamo due algoritmi che, in Java, costruiscono lunghe stringhe. Il nostro obiettivo sarà quello di avere un metodo, con una firma simile a `repeat('*', 40)`, che genera una stringa costituita da 40 asterischi: "*****
*****".

Il primo algoritmo che prendiamo in esame esegue ripetute concatenazioni tra stringhe, usando l'operatore `+`, e viene implementato nel metodo `repeat1`, nel Codice 4.2. Il secondo algoritmo si basa sulla classe di libreria `StringBuilder` (presentata nel Paragrafo 1.3) e viene implementato nel metodo `repeat2`, di nuovo nel Codice 4.2.

Codice 4.2: Due algoritmi che costruiscono una stringa ripetendo uno stesso carattere.

```

1  /** Usa la concatenazione per costruire una stringa con n copie del carattere c. */
2  public static String repeat1(char c, int n) {
3      String answer = "";
4      for (int j=0; j < n; j++)
5          answer += c;
6      return answer;
7  }
8
9  /** Usa StringBuilder per costruire una stringa con n copie del carattere c. */
10 public static String repeat2(char c, int n) {
11     StringBuilder sb = new StringBuilder();

```

```

    for (int j=0; j < n; j++)
        sb.append(c);
    return sb.toString();
}

```

Durante l'esperimento, abbiamo usato `System.currentTimeMillis()`, come nel Codice 4.1, per misurare l'efficienza tanto di `repeat1` quanto di `repeat2` nella generazione di stringhe di grandi dimensioni. Abbiamo costruito stringhe di lunghezza crescente, per individuare la relazione esistente tra il tempo d'esecuzione e la lunghezza della stringa generata. I risultati dei nostri esperimenti sono riportati nella Tabella 4.1 e la Figura 4.1 riporta il relativo grafico, in scala logaritmica su entrambi gli assi (*log-log*).

Tabella 4.1: Risultati dell'esperimento di misura del tempo di esecuzione dei metodi `repeat1` e `repeat2` presentati nel Codice 4.2

N	<code>repeat1</code> (in ms)	<code>repeat2</code> (in ms)
50000	2884	1
100000	7437	1
200000	39158	2
400000	170173	3
800000	690836	7
1600000	2874968	13
3200000	12809631	28
6400000	59594275	58
12800000	265696421	135

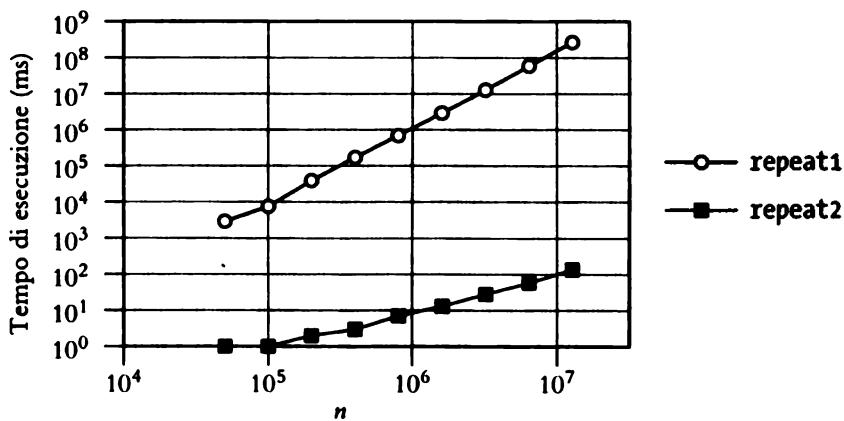


Figura 4.1: Grafico che riporta i risultati dell'esperimento relativo al Codice 4.2, in scala logaritmica su entrambi gli assi. Le pendenze divergenti evidenziano un andamento ben diverso nella crescita dei tempi d'esecuzione.

Il risultato più eclatante di questi esperimenti è la velocità decisamente più elevata dell'algoritmo `repeat2` rispetto a `repeat1`. Mentre `repeat1` richiede più di 3 giorni per costruire una stringa con 12.8 milioni di caratteri, `repeat2` è stato in grado di raggiungere lo stesso risultato in una frazione di secondo. Esaminando il grafico, poi, si possono dedurre le relazioni esistenti tra il tempo d'esecuzione di ciascun algoritmo e la dimensione n del problema da risolvere. Al raddoppiare del valore di n , il tempo d'esecuzione di `repeat1` diventa tipicamente più del quadruplo, mentre quello di `repeat2` approssimativamente raddoppia.

Dificoltà nelle analisi sperimentali

Anche se le analisi sperimentali del tempo d'esecuzione sono utili, in particolar modo quando si mette a punto il codice per fargli raggiungere una qualità adeguata ad entrare in produzione, il loro utilizzo nell'analisi degli algoritmi presenta tre importanti limitazioni:

- È difficile mettere direttamente a confronto i tempi di esecuzione rilevati sperimentalmente per due algoritmi, a meno che gli esperimenti non vengano effettuati nel medesimo ambiente hardware e software.
- Gli esperimenti possono essere eseguiti soltanto su un insieme limitato di dati in ingresso, per cui non possono tener conto del tempo di esecuzione di tutti i casi non previsti dagli esperimenti stessi (e alcuni di questi possono essere importanti).
- Per poter essere eseguito e analizzato sperimentalmente, l'algoritmo in esame deve essere interamente implementato.

Quest'ultimo requisito è lo svantaggio più rilevante che deriva dall'utilizzo delle analisi sperimentali. Nelle prime fasi dello sviluppo di un progetto, quando si prendono in considerazione varie strutture dati e algoritmi per prendere una decisione, sarebbe folle dedicare tanto tempo all'implementazione di un approccio che potrebbe facilmente essere dichiarato inferiore da un'analisi di alto livello.

4.1.1 Superare le analisi sperimentali

Il nostro obiettivo è lo sviluppo di un approccio che consenta di analizzare l'efficienza degli algoritmi e che:

1. Consenta di valutare l'efficienza relativa di due algoritmi in modo indipendente dall'ambiente hardware e software
2. Operi analizzando una descrizione di alto livello dell'algoritmo, senza che sia necessario implementarlo.
3. Tenga in considerazione tutti i possibili insiemi di dati in ingresso.

Contare le operazioni elementari

Per analizzare il tempo di esecuzione di un algoritmo senza eseguire esperimenti, effettuiamo l'analisi direttamente su una sua descrizione di alto livello (sotto forma di una effettiva sezione di codice o di uno pseudocodice indipendente dai linguaggi di programmazione). A questo scopo, definiamo un insieme di *operazioni elementari* o *primitive*:

- Assegnare un valore a una variabile
- Seguire un riferimento a un oggetto

- Eseguire un'operazione aritmetica (ad esempio, sommare due numeri)
- Confrontare due numeri
- Accedere a un singolo elemento di un array usando il suo indice
- Invocare un metodo
- Restituire un valore al termine dell'esecuzione di un metodo

Dal punto di vista formale, un'operazione elementare corrisponde a un'istruzione di basso livello il cui tempo di esecuzione sia costante e, almeno in teoria, dovrebbe essere un tipo di operazione che viene eseguita direttamente come una singola istruzione hardware, anche se molte delle nostre operazioni elementari possono, in realtà, essere tradotte in un piccolo numero di istruzioni hardware. Invece di cercare di determinare l'effettivo tempo d'esecuzione di ciascuna operazione elementare, conteremo semplicemente quante di esse vengono eseguite, usando tale conteggio, t , come misura del tempo d'esecuzione dell'algoritmo.

Questo conteggio di operazioni sarà correlato all'effettivo tempo d'esecuzione su uno specifico computer, perché ciascuna operazione elementare corrisponde a un numero costante di istruzioni hardware, e il numero di diverse operazioni elementari è prefissato. L'ipotesi implicita su cui si basa questo approccio è che i tempi di esecuzione di operazioni elementari diverse siano abbastanza simili. Di conseguenza, il numero t di operazioni elementari eseguite da un algoritmo sarà proporzionale all'effettivo tempo di esecuzione dell'algoritmo stesso.

Contare le operazioni in funzione della dimensione dei dati

Per individuare la velocità di crescita del tempo di esecuzione di un algoritmo in funzione della dimensione n dei dati da elaborare, associeremo a ogni algoritmo una funzione, $f(n)$, che esprima il numero di operazioni elementari eseguite dall'algoritmo in funzione, appunto, della dimensione n . Il Paragrafo 4.2 presenterà le sette funzioni che vengono prodotte più frequentemente da questa analisi, mentre il Paragrafo 4.3 discuterà un'infrastruttura matematica per confrontare diverse funzioni.

Porre l'attenzione sul caso peggiore

L'esecuzione di un algoritmo può essere rapida con alcuni dati in ingresso e più lenta con altri, pur avendo la stessa dimensione, per cui potremmo voler esprimere il tempo d'esecuzione di un algoritmo in funzione della dimensione dei dati in ingresso calcolando il suo valore medio su tutti i possibili casi. Sfortunatamente, tale analisi *del caso medio* (*average-case analysis*) è solitamente piuttosto complessa e richiede la definizione di una distribuzione di probabilità per gli insiemi dei dati in ingresso, che, spesso, è un problema di difficile soluzione. La Figura 4.2 mostra schematicamente come, in relazione alla distribuzione degli ingressi, il tempo di esecuzione di un algoritmo può collocarsi in qualunque punto tra il tempo richiesto nel caso peggiore e quello richiesto nel caso migliore. Ad esempio, cosa succede se i dati in ingresso sono effettivamente tutti del tipo "A" o "D"?

Solitamente per un'analisi del caso medio è richiesto il calcolo del tempo d'esecuzione *atteso* sulla base di una data distribuzione dei dati in ingresso, cosa che di solito richiede un uso sofisticato della teoria della probabilità. Quindi, in questo libro, tranne dove diversamente specificato, caratterizzeremo sempre i tempi d'esecuzione degli algoritmi nel *caso peggiore*, in funzione della dimensione, n , dei dati in ingresso.

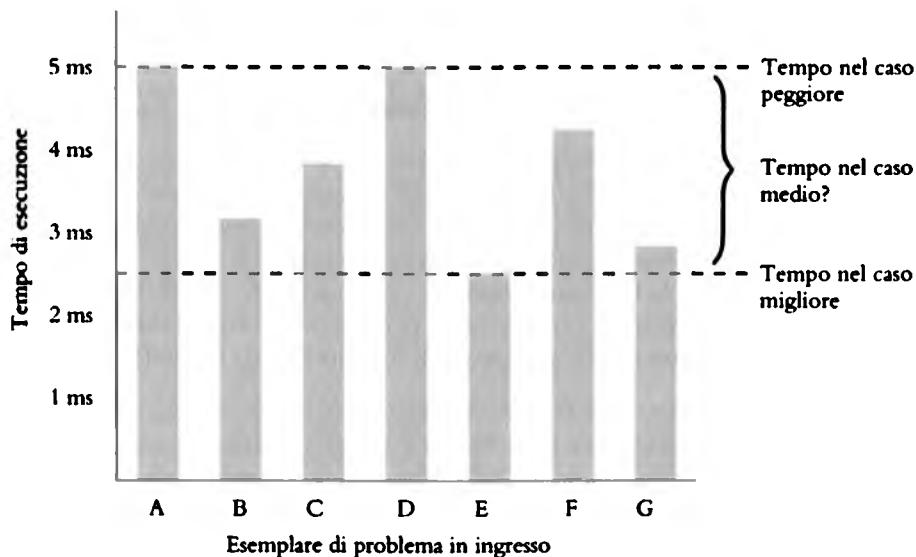


Figura 4.2: Differenza tra i tempi d'esecuzione nel caso migliore e nel caso peggiore.
Ogni barra rappresenta il tempo d'esecuzione di un certo algoritmo eseguito su un diverso insieme di dati in ingresso.

L'analisi nel caso peggiore è molto più facile dell'analisi nel caso medio, perché richiede soltanto di identificare quale sia, appunto, il caso peggiore per i dati in ingresso, cosa che solitamente non è difficile. Inoltre, questo approccio porta spesso alla progettazione di algoritmi migliori: facendo in modo che un algoritmo venga dichiarato vincente quando ha buone prestazioni nel caso peggiore, si otterranno algoritmi che hanno buone prestazioni con *qualsiasi* dato in ingresso. In pratica, progettando per il caso peggiore si tenderà a "rafforzare la muscolatura" degli algoritmi, un po' come una stella dell'atletica che si alleni correndo sempre in salita.

4.2 Le sette funzioni usate in questo libro

In questo paragrafo analizzeremo brevemente le sette principali funzioni utilizzate nell'analisi di algoritmi, che saranno le sole utilizzate per la maggior parte delle analisi condotte in questo libro (se un paragrafo usa una funzione diversa da queste sette, lo contrassegneremo con un asterisco, evidenziando il fatto che si tratta di un argomento facoltativo). Oltre a queste sette funzioni fondamentali, l'Appendice del libro contiene un elenco di altre utili proprietà matematiche che si applicano all'analisi delle strutture dati e degli algoritmi.

La funzione costante

La funzione più semplice che possiamo immaginare è la **funzione costante**:

$$f(n) = c,$$

con c costante prefissata, come $c = 5$, $c = 27$ o $c = 2^{10}$. Quindi, per qualsiasi valore dell'argomento n , la funzione costante $f(n)$ assegna il valore c . In altre parole, indipendentemente dal valore di n , che è ininfluente, il valore di $f(n)$ sarà sempre uguale al valore costante c .

Dal momento che siamo principalmente interessati alle funzioni con valore intero, la funzione costante fondamentale è $g(n) = 1$, e proprio questa è la funzione costante tipicamente utilizzata nel libro. Si noti che qualunque altra funzione costante, come $f(n) = c$, può essere scritta come una costante c moltiplicata per $g(n)$, cioè $f(n) = cg(n)$.

Per quanto semplice, la funzione costante è comunque utile nell'analisi di algoritmi, perché caratterizza il numero di passi necessari per compiere un'operazione elementare, come la somma di due numeri, l'assegnazione di un valore a una variabile o il confronto tra due numeri.

La funzione logaritmo

Uno degli aspetti interessanti e per certi versi anche sorprendente dell'analisi delle strutture dati e degli algoritmi è la presenza assai diffusa della **funzione logaritmo**, $f(n) = \log_b n$, con b costante e $b > 1$. Questa funzione è definita come la funzione inversa dell'elevamento a potenza, in questo modo:

$$x = \log_b n \text{ se e solo se } b^x = n.$$

Il valore di b prende il nome di **base** del logaritmo. Si noti che dalla definizione precedente discende il fatto che, per qualunque base $b > 0$, si ha $\log_b 1 = 0$.

La base più frequentemente utilizzata in informatica per la funzione logaritmo è 2, perché i calcolatori memorizzano i numeri interi in formato binario. Tale base è, in effetti, così frequente che tipicamente la ometteremo dalla notazione tutte le volte che il suo valore è 2; quindi, per noi:

$$\log n = \log_2 n.$$

Notiamo anche che la maggior parte delle calcolatrici dispone di un pulsante **LOG**, che, però, viene solitamente utilizzato per calcolare il logaritmo in base 10, non in base 2.

Per calcolare il valore esatto della funzione logaritmo per un numero intero n occorrono tecniche di calcolo numerico, ma possiamo evitarlo e usare un'approssimazione che è sufficiente per i nostri scopi. Ricordiamo che, dato un numero reale x , il valore della funzione **parte intera superiore**, che indichiamo con $\lceil x \rceil$, è uguale al minimo numero intero non minore di x (in inglese tale funzione viene chiamata *ceiling*, cioè "soffitto"). La parte intera superiore può essere considerata un'approssimazione intera di x , dato che si ha $x \leq \lceil x \rceil < x + 1$. Dato un numero intero positivo, n , possiamo dividere ripetutamente n per b , fino a quando otteniamo un numero ≤ 1 : il numero di divisioni eseguite è uguale a $\lceil \log_b n \rceil$. Vediamo alcuni esempi del calcolo di $\lceil \log_b n \rceil$ mediante divisioni ripetute:

- $\lceil \log_3 27 \rceil = 3$, perché $((27/3)/3)/3 = 1$
- $\lceil \log_4 64 \rceil = 3$, perché $((64/4)/4)/4 = 1$
- $\lceil \log_2 12 \rceil = 4$, perché $((12/2)/2)/2 = 0.75 \leq 1$

La proposizione seguente descrive alcune identità importanti che riguardano i logaritmi e sono valide per qualsiasi base maggiore di 1.

Proposizione 4.1 (Regole del logaritmo): *Dati i numeri reali $a > 0$, $b > 1$, $c > 0$ e $d > 1$, abbiamo:*

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c \log_b a$
4. $\log_b a = \log_d a / \log_d b$
5. $b^{\log_b a} = a^{\log_d b}$

Per convenzione, la notazione $\log n^b$ priva di parentesi rappresenta il valore $\log(n^b)$. Inoltre, si usa una notazione abbreviata, $\log^b n$, per indicare la quantità $(\log n)^b$, cioè il risultato del logaritmo elevato a potenza.

Le identità appena viste possono essere derivate dalle regole inverse dell'elevamento a potenza che vedremo nella Proposizione 4.4; le illustriamo con alcuni esempi.

Esempio 4.2: Elenchiamo qui alcune applicazioni interessanti delle regole del logaritmo riportate nella Proposizione 4.1 (usando la solita convenzione che prevede di omettere la base del logaritmo quando è uguale a 2).

- $\log(2n) = \log 2 + \log n = 1 + \log n$, per la regola 1
- $\log(n/2) = \log n - \log 2 = \log n - 1$, per la regola 2
- $\log n^3 = 3 \log n$, per la regola 3
- $\log 2^n = n \log 2 = n \cdot 1 = n$, per la regola 3
- $\log_4 n = (\log n) / \log 4 = (\log n) / 2$, per la regola 4
- $2^{\log n} = n^{\log 2} = n^1 = n$, per la regola 5

Dal punto di vista pratico, osserviamo che la regola 4 ci consente di calcolare il logaritmo in base 2 usando una calcolatrice che abbia il pulsante LOG per calcolare il logaritmo in base 10, perché:

$$\log_2 n = \text{LOG } n / \text{LOG } 2.$$

La funzione lineare

Un'altra funzione tanto semplice quanto importante è la *funzione lineare*:

$$f(n) = n.$$

La funzione lineare f assegna, quindi, a un valore n il valore n stesso.

Questa funzione emerge nell'analisi degli algoritmi ogniqualvolta si ha a che fare con un'unica operazione elementare per ciascuno di n elementi. Ad esempio, confrontare un numero x con ciascun elemento di un array di dimensione n richiede n confronti. La funzione lineare rappresenta anche il tempo d'esecuzione migliore che possiamo sperare

di raggiungere con qualunque algoritmo che elabori singolarmente n oggetti che non si trovino già all'interno della memoria del calcolatore, perché l'acquisizione di n oggetti richiede, già di per se stessa, n operazioni.

La funzione $N \log N$

La prossima funzione di cui parliamo in questo paragrafo è la *funzione $n \log n$* :

$$f(n) = n \log n,$$

che, al valore n , assegna il valore pari a n volte il logaritmo in base 2 di n . Questa funzione cresce appena un po' più rapidamente della funzione lineare e molto meno rapidamente della funzione quadratica; quindi, un algoritmo con tempo d'esecuzione proporzionale a $n \log n$ sarà decisamente preferibile rispetto a un algoritmo con tempo d'esecuzione quadratico. Vedremo parecchi importanti algoritmi che sono caratterizzati da un tempo d'esecuzione proporzionale alla funzione $n \log n$. Ad esempio, i più veloci algoritmi che possano esistere per ordinare n valori arbitrari richiedono un tempo d'esecuzione proporzionale proprio a $n \log n$.

La funzione quadratica

Un'altra funzione che compare spesso nell'analisi di algoritmi è la *funzione quadratica*:

$$f(n) = n^2,$$

che, al valore n , assegna il valore pari al prodotto di n per se stesso (in altre parole, " n al quadrato").

Il motivo principale che giustifica la comparsa della funzione quadratica nell'analisi degli algoritmi sta nel fatto che molti di essi hanno cicli annidati, dove il ciclo interno esegue un numero lineare di operazioni e il ciclo esterno viene eseguito un numero lineare di volte: in tali casi, quindi, l'algoritmo esegue $n \cdot n = n^2$ operazioni.

I cicli annidati e la funzione quadratica

La funzione quadratica può emergere anche nel contesto di cicli annidati nei casi in cui la prima iterazione del ciclo esegue un'operazione, la seconda iterazione esegue due operazioni, la terza tre operazioni, e così via. Il numero totale di operazioni è, quindi:

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n.$$

In altre parole, questo è il numero totale di operazioni che sarà eseguito dal ciclo annidato se il numero di operazioni eseguite dal ciclo interno aumenta di un'unità dopo ogni iterazione del ciclo esterno. Questa somma ha anche una storia interessante.

Nel 1787, un insegnante tedesco decise di tenere occupati i suoi scolari di 9 e 10 anni facendo loro sommare i numeri interi da 1 a 100, ma uno di loro asserrì quasi immediatamente di avere la risposta! L'insegnante si insospettì, perché lo scolaro aveva solamente la risposta scritta sulla sua lavagnetta, senza alcun calcolo. Ma la risposta, 5050, era corretta e lo studente, Carl Gauss, si rivelò poi come uno dei più grandi matema-

tità del suo tempo. È lecito ipotizzare che il giovane Gauss abbia usato la seguente identità.

Proposizione 4.3: *Per qualsiasi numero intero $n \geq 1$, si ha:*

$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

Nella Figura 4.3 forniamo due dimostrazioni “grafiche” della Proposizione 4.3.

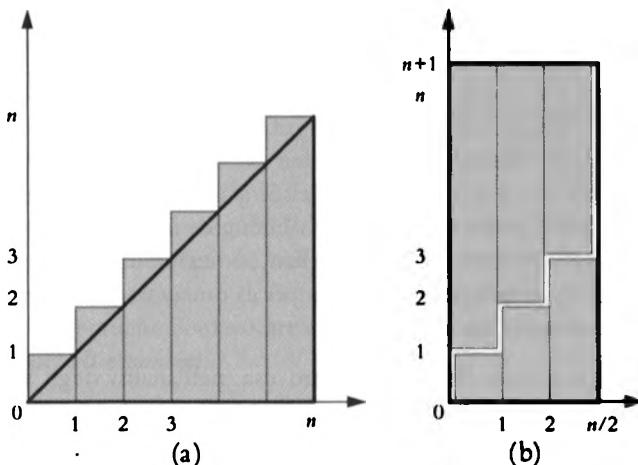


Figura 4.3: Giustificazioni grafiche della Proposizione 4.3 Entrambe le figure visualizzano il significato dell’identità in termini di area totale coperta da n rettangoli di larghezza unitaria con altezze uguali a $1, 2, \dots, n$. Nel caso (a) si vede come i rettangoli coprano un grande triangolo di area $n^2/2$ (la base è n , come l’altezza) oltre a n piccoli triangoli, ciascuno dei quali ha area $1/2$ (la base è 1, come l’altezza). Nel caso (b), che si può applicare solamente quando n è un numero pari, si vede come i rettangoli coprano un grande rettangolo avente base $n/2$ e altezza $n+1$.

Dalla Proposizione 4.3 abbiamo imparato una lezione: se eseguiamo un algoritmo che abbia cicli annidati tali che le operazioni del ciclo interno aumentano di un’unità ogni volta, allora il numero totale di operazioni è quadratico in funzione del numero di esecuzioni, n , del ciclo esterno. Per essere precisi, il numero di operazioni è $n^2/2 + n/2$, quindi è maggiore della metà del numero di operazioni eseguite da un algoritmo che usa n operazioni per ogni iterazione del ciclo interno, ma l’ordine di grandezza è comunque quadratico in funzione di n .

La funzione cubica e altre polinomiali

Procedendo con la trattazione di funzioni che siano potenze di n , prendiamo in esame la **funzione cubica**:

$$f(n) = n^3,$$

che, al valore n , assegna il valore pari al prodotto di n per se stesso tre volte (in altre parole " n al cubo").

La funzione cubica si ritrova meno frequentemente nel contesto dell'analisi degli algoritmi rispetto alle funzioni costante, lineare e quadratica già menzionate, ma di tanto in tanto la si riscontra.

Funzioni polinomiali

Le funzioni lineare, quadratica e cubica possono essere considerate come appartenenti a una più grande categoria di funzioni, le *polinomiali*. Una funzione polinomiale ha questa forma:

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d,$$

dove a_0, a_1, \dots, a_d sono costanti, chiamate *coefficientsi* della polinomiale, con $a_d \neq 0$. Il numero intero d , che indica l'esponente più elevato nella polinomiale, si chiama *grado* del polinomio.

Ad esempio, le funzioni seguenti sono tutte polinomiali:

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$

Potremmo, quindi, affermare che questo libro usa, nell'analisi degli algoritmi, soltanto quattro funzioni principali, ma continueremo a parlare di sette funzioni, perché le funzioni costante, lineare e quadratica sono troppo importanti per essere raggruppate assieme alle altre polinomiali. I tempi d'esecuzione che sono polinomiali con un grado basso sono, in generale, migliori dei tempi d'esecuzione che sono polinomiali con un grado più elevato.

Sommatorie

La *sommatoria* è una notazione che compare ripetutamente nell'analisi delle strutture dati e degli algoritmi, ed è così definita:

$$\sum_{i=a}^b f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

dove a e b sono numeri interi, con $a \leq b$. Le sommatorie compaiono nell'analisi di algoritmi e strutture dati perché i tempi di esecuzione dei cicli danno luogo a sommatorie in modo assai naturale.

Usando una sommatoria, possiamo riscrivere la formula della Proposizione 4.3 in questo modo:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Analogamente, possiamo scrivere in questo modo una funzione $f(n)$ polinomiale di grado d con coefficienti a_0, \dots, a_d :

$$f(n) = \sum_{i=0}^d a_i n^i$$

Quindi, la notazione di sommatoria ci mette a disposizione un modo abbreviato per esprimere somme di termini che manifestano una struttura regolare.

La funzione esponenziale

L'ultima funzione che presentiamo, tra quelle utilizzate nell'analisi di algoritmi, è la **funzione esponenziale**:

$$f(n) = b^n,$$

dove b è una costante positiva, chiamata **base**, e l'argomento della funzione, n , funge da **esponente**. Questa funzione, quindi, assegna al valore n il valore che si ottiene moltiplicando la base b per se stessa n volte. Come visto nel caso della funzione logaritmo, la base più frequente per la funzione esponenziale, nel contesto dell'analisi di algoritmi, è $b = 2$. Ad esempio, una parola binaria di n bit può rappresentare tutti i numeri interi non negativi minori di 2^n . Un ciclo che inizi eseguendo una sola operazione e poi raddoppi il numero di operazioni eseguite a ogni iterazione, nella sua n -esima iterazione eseguirà 2^n operazioni.

A volte, comunque, ci troviamo di fronte a esponenti diversi da n , per cui è utile conoscere alcune regole pratiche per lavorare con gli esponenti. In particolare, sono piuttosto utili queste **regole per gli esponenti**.

Proposizione 4.4 (Regole per gli esponenti): *Dati i numeri interi positivi a , b e c , abbiamo:*

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

Ad esempio:

- $256 = 16^2 = (2^4)^2 = 2^{4 \cdot 2} = 2^8 = 256$ (Regola 1 degli esponenti)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$ (Regola 2 degli esponenti)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$ (Regola 3 degli esponenti)

La funzione esponenziale può essere estesa al caso di esponente che sia una frazione o un numero reale, o anche un numero negativo, in questo modo. Dato un numero intero positivo k , definiamo $b^{1/k}$ come la radice k -esima di b , cioè il numero r tale che $r^k = b$. Ad esempio, $25^{1/2} = 5$, perché $5^2 = 25$. Analogamente, $27^{1/3} = 3$ e $16^{1/4} = 2$. Questo approccio ci consente di definire qualunque potenza il cui esponente possa essere espresso mediante una frazione, perché $b^{a/c} = (b^a)^{1/c}$, in base alla Regola 1 degli esponenti. Per esempio, $9^{3/2} = (9^3)^{1/2} = 729^{1/2} = 27$. Di conseguenza, $b^{a/c}$ è, in realtà, la radice c -esima della potenza intera b^a .

Possiamo estendere ulteriormente la funzione esponenziale, dando un significato a b^x per qualunque numero reale x : basta calcolare il valore a cui converge la serie di numeri aventi

la forma $b^{a/c}$ per frazioni a/c che si avvicinino sempre di più a x . Qualunque numero reale x può essere approssimato con precisione arbitraria da una frazione a/c , quindi possiamo usare tale frazione come esponente di b per avvicinarci arbitrariamente a b^x . Ad esempio, il numero 2^x è ben definito. Infine, dato un esponente negativo d , definiamo $b^d = 1/b^{-d}$, che corrisponde all'applicazione della Regola 3 degli esponenti con $a = 0$ e $c = -d$. Ad esempio, $2^{-3} = 1/2^3 = 1/8$.

Somme geometriche

Immaginiamo di avere un ciclo in cui ciascuna iterazione richiede un tempo maggiore di quello richiesto dalla precedente per un fattore moltiplicativo. Ecco una proposizione che consente di analizzare tale ciclo.

Proposizione 4.5: *Per qualsiasi numero intero $n \geq 0$ e qualsiasi numero reale a tale che $a > 0$ e $a \neq 1$, consideriamo la sommatoria:*

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \cdots + a^n$$

(ricordando che $a^0 = 1$ se $a > 0$). Questa sommatoria è uguale a:

$$\frac{a^{n+1} - 1}{a - 1}$$

Le sommatorie presentate nella Proposizione 4.5 prendono il nome di sommatorie *geometriche*, perché ciascun termine è geometricamente maggiore del precedente, se $a > 1$. Ad esempio, chiunque lavori nel settore informatico sa bene che:

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1,$$

perché questo è il massimo numero intero che può essere rappresentato, senza segno, con la notazione binaria usando n bit.

4.2.1 Confrontare velocità di crescita

Per riassumere, la Tabella 4.2 mostra, in ordine crescente, le sette funzioni più utilizzate nell'analisi di algoritmi.

Tabella 4.2: Le sette funzioni più utilizzate nell'analisi di algoritmi. Ricordiamo anche qui che $\log n = \log_2 n$. Inoltre, a è una costante maggiore di 1.

costante	logaritmica	lineare	$n \log n$	quadratica	cubica	esponenziale
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

In una situazione ideale, vorremmo che le operazioni sulle strutture dati venissero eseguite in tempi proporzionali alla funzione costante o alla funzione logaritmo, così come vorrem-

che i nostri algoritmi richiedessero un tempo d'esecuzione lineare o proporzionale a $\log n$. Gli algoritmi con tempi d'esecuzione quadratici o cubici sono meno utilizzabili, gli algoritmi con tempi esponenziali sono addirittura inutilizzabili per dati in ingresso la cui dimensione non sia veramente piccola. La Figura 4.4 riporta, sotto forma di grafico cartesiano, l'andamento delle sette funzioni.

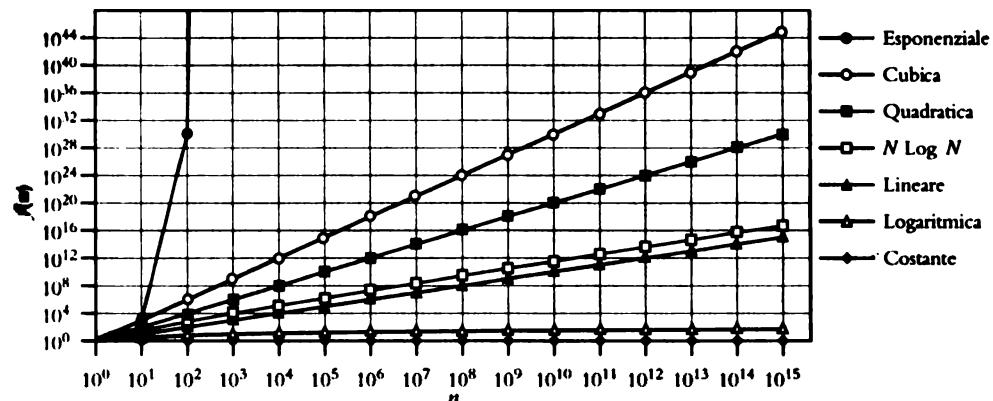


Figura 4.4: Velocità di crescita delle sette funzioni fondamentali usate nell'analisi di algoritmi. Per la funzione esponenziale abbiamo usato la base $a = 2$. Le funzioni sono rappresentate in un diagramma log-log (cioè in scala logaritmica su entrambi gli assi) per poter confrontare soprattutto le pendenze delle curve. Anche con questo accorgimento, la funzione esponenziale cresce troppo rapidamente per poter visualizzare nel grafico tutti i suoi valori.

Le funzioni “parte intera” inferiore e superiore (*ceiling* e *floor*)

Parlando di logaritmi, abbiamo osservato che il valore della funzione logaritmo, in generale, non è un numero intero, tuttavia il tempo d'esecuzione di un algoritmo viene solitamente espresso mediante una quantità intera, perché è rappresentato dal numero di operazioni eseguite. Di conseguenza, a volte l'analisi di un algoritmo richiede l'utilizzo della **funzione parte intera (inferiore)** (o, in inglese, *floor*, che significa “pavimento”, cioè qualcosa che “sta sotto”) e della **funzione parte intera superiore** (o, in inglese, *ceiling*, che significa “soffitto”), cioè qualcosa che “sta sopra”), che sono rispettivamente definite in questo modo:

- $\lfloor x \rfloor$ = il massimo numero intero non maggiore di x (ad esempio, $\lfloor 3.7 \rfloor = 3$)
- $\lceil x \rceil$ = il minimo numero intero non minore di x (ad esempio, $\lceil 5.2 \rceil = 6$)

4.3 Analisi asintotica

Nell'analisi degli algoritmi ci concentriamo sulla velocità di crescita del tempo d'esecuzione in funzione della dimensione, n , dei dati in ingresso, seguendo un approccio che cerca di cogliere l'*andamento* di tale tempo. Ad esempio, spesso è sufficiente sapere che il tempo d'esecuzione di un algoritmo *cresce proporzionalmente a n*.

Analizziamo gli algoritmi usando una notazione matematica che caratterizza le funzioni ignorando fattori costanti. Più precisamente, caratterizziamo i tempi di esecuzione degli algoritmi usando funzioni che mettono in corrispondenza la dimensione n dei dati in ingresso con i valori derivanti dal fattore principale che determina la velocità di crescita in funzione di n . Questo approccio è conseguenza del fatto che ogni passo di una descrizione dell'algoritmo mediante pseudocodice o della sua implementazione in un linguaggio di alto livello può corrispondere a un piccolo numero di operazioni elementari. Quindi, possiamo effettuare l'analisi di un algoritmo stimando il numero di operazioni elementari eseguite a meno di un fattore costante, invece di addentrarci in analisi dipendenti dal linguaggio o dall'hardware, che volessero valutare il numero esatto di operazioni eseguite dal calcolatore.

4.3.1 La notazione "O-grande"

Siano $f(n)$ e $g(n)$ funzioni che mettono in corrispondenza numeri interi positivi con numeri reali positivi. Diciamo che $f(n)$ è $O(g(n))$ se esiste una costante reale $c > 0$ e una costante intera $n_0 \geq 1$ tali che:

$$f(n) \leq c \cdot g(n), \text{ per } n \geq n_0.$$

Questa definizione viene spesso chiamata notazione "O-grande" (Big-Oh, in inglese), perché a volte si legge come " $f(n)$ è O-grande di $g(n)$ ". La Figura 4.5 illustra la definizione.

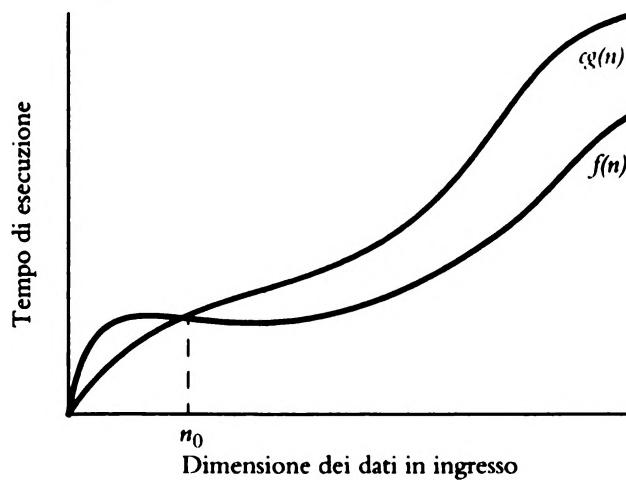


Figura 4.5: Descrizione grafica della notazione "O-grande". La funzione $f(n)$ è $O(g(n))$, perché $f(n) \leq c \cdot g(n)$ quando $n \geq n_0$.

Esempio 4.6: La funzione $8n + 5$ è $O(n)$.

Dimostrazione: In base alla definizione di O-grande, dobbiamo trovare una costante reale $c > 0$ e una costante intera $n_0 \geq 1$ tali che sia $8n + 5 \leq cn$ per ogni numero intero $n \geq n_0$. Non è difficile osservare che $c = 9$ e $n_0 = 5$ sono una coppia di valori possibili, anche

~~In realtà, ci sono infinite coppie adatte, perché si tratta di gestire un compromesso tra c e n₀.~~ Ad esempio, si potrebbero usare le costanti $c = 13$ e $n_0 = 1$. ■

La notazione O-grande ci consente di affermare che una funzione $f(n)$ è "minore di o uguale a" (cioè "non maggiore di") un'altra funzione $g(n)$ a meno di un fattore costante, in senso asintotico, cioè per n che cresce verso l'infinito. Questa possibilità deriva dal fatto che la definizione usa " \leq " per fare il confronto tra $f(n)$ e una costante c moltiplicata per $g(n)$, nel caso in cui sia asintoticamente $n \geq n_0$. Nonostante ciò, non è ritenuto corretto dire che " $f(n) \leq O(g(n))$ ", perché la notazione O-grande comprende già al proprio interno il concetto di "minore di o uguale a". Analogamente, non è del tutto corretto scrivere che " $f(n) = O(g(n))$ ", anche se lo si fa spesso, perché il significato consueto del simbolo " $=$ " è quello di una relazione di equivalenza, ma non ha alcun senso scrivere l'enunciato simmetrico, " $O(g(n)) = f(n)$ ". È decisamente preferibile dire che " $f(n) \in O(g(n))$ ".

In alternativa, possiamo dire che " $f(n)$ è dell'ordine di $g(n)$ ". Per chi abbia una vera passione per la matematica, è anche corretto affermare che " $f(n) \in O(g(n))$ ", perché, in senso tecnico, la notazione O-grande individua un intero insieme di funzioni. In questo libro abbiamo deciso di presentare gli enunciati che usano O-grande nella forma " $f(n) \in O(g(n))$ ". Anche con questa scelta, rimane una significativa libertà nell'uso di operazioni aritmetiche che coinvolgono la notazione O-grande: a questa libertà consegue un'analogia responsabilità.

Alcune proprietà della notazione O-grande

La notazione O-grande ci consente di ignorare i fattori costanti e i termini di grado inferiore, ponendo l'attenzione sulle componenti di una funzione che influenzano in modo prevalente la sua crescita.

Esempio 4.7: $5n^4 + 3n^3 + 2n^2 + 4n + 1 \in O(n^4)$.

Dimostrazione: Si osservi che $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)n^4 = cn^4$, per $c = 15$, quando $n \geq n_0 = 1$. ■

In modo analogo possiamo, in effetti, caratterizzare l'andamento di qualunque funzione polinomiale.

Proposizione 4.8: Se $f(n)$ è una funzione polinomiale di grado d , cioè se:

$$f(n) = a_0 + a_1n + \dots + a_dn^d,$$

e $a_d > 0$, allora $f(n) \in O(n^d)$.

Dimostrazione: Si osservi che, per $n \geq 1$, valgono le relazioni $1 \leq n \leq n^2 \leq \dots \leq n^d$, per cui:

$$a_0 + a_1n + a_2n^2 + \dots + a_dn^d \leq (|a_0| + |a_1| + |a_2| + \dots + |a_d|)n^d.$$

Dimostriamo, quindi, che $f(n) \in O(n^d)$ definendo $c = |a_0| + |a_1| + \dots + |a_d|$ e $n_0 = 1$. ■

Di conseguenza, il termine di grado massimo di una funzione polinomiale è quello che ne determina l'andamento asintotico. Negli esercizi prenderemo in esame alcune ulteriori proprietà della notazione O-grande, ma vediamo ancora qualche esempio anche qui, concentrandoci su combinazioni delle sette funzioni fondamentali usate nell'analisi di algoritmi e sfruttando la seguente proprietà matematica: $\log n \leq n$ per ogni $n \geq 1$.

Esempio 4.9: $5n^2 + 3n \log n + 2n + 5$ è $O(n^2)$.

Dimostrazione: $5n^2 + 3n \log n + 2n + 5 \leq (5 + 3 + 2 + 5) n^2 = cn^2$, per $c = 15$, quando $n \geq n_0 = 1$. ■

Esempio 4.10: $20n^3 + 10n \log n + 5$ è $O(n^3)$.

Dimostrazione: $20n^3 + 10n \log n + 5 \leq 35n^3$ per $n \geq 1$. ■

Esempio 4.11: $3 \log n + 2$ è $O(\log n)$.

Dimostrazione: $3 \log n + 2 \leq 5 \log n$ per $n \geq 2$. Si noti che $\log n$ vale zero per $n = 1$; per tale motivo in questo caso usiamo $n \geq n_0 = 2$. ■

Esempio 4.12: 2^{n+2} è $O(2^n)$.

Dimostrazione: $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$; quindi, in questo caso possiamo prendere $c = 4$ e $n_0 = 1$. ■

Esempio 4.13: $2n + 100 \log n$ è $O(n)$.

Dimostrazione: $2n + 100 \log n \leq 102n$ per $n \geq n_0 = 1$; quindi, in questo caso possiamo prendere $c = 102$. ■

Caratterizzare le funzioni nel modo più stringente

In generale, dovremmo usare la notazione O-grande per caratterizzare una funzione nel modo più stringente possibile. Anche se è certamente vero che la funzione $f(n) = 4n^3 + 3n^2$ è $O(n^5)$ o anche $O(n^4)$, è più preciso dire che $f(n)$ è $O(n^3)$. Per fare un'analogia, considerate uno scenario nel quale un viaggiatore affamato si trova lungo una strada di campagna, dove incontra un contadino che torna a casa a piedi dal supermercato. Se il viaggiatore chiede al contadino per quanto tempo dovrà guidare prima di poter trovare del cibo, certamente il contadino non mente se afferma "sicuramente meno di 12 ore", ma è molto più preciso (e utile) se dice "c'è un supermercato a pochi minuti da qui". Anche con la notazione O-grande dovete fare ogni sforzo possibile per raccontare l'intera verità.

Si ritiene che sia anche poco elegante inserire nella notazione O-grande fattori costanti e termini di grado non massimo. Ad esempio, solitamente non si dice (nonostante sia certamente vero) che la funzione $2n^2$ è $O(4n^2 + 6n \log n)$. Dobbiamo sforzarci di

descrivere la funzione in notazione O-grande nella sua forma più semplice o ai minimi errori.

Le sette funzioni elencate nel Paragrafo 4.2 sono le più comuni ad essere utilizzate insieme alla notazione O-grande per caratterizzare gli algoritmi in merito al tempo di esecuzione e allo spazio utilizzato. In effetti, solitamente usiamo proprio i nomi di queste funzioni per identificare il tempo d'esecuzione degli algoritmi che esse caratterizzano, per cui, ad esempio, diremo che un algoritmo il cui tempo d'esecuzione nel caso peggiore è $4n^2 + n \log n$ è un algoritmo *tempo-quadratico*, perché viene eseguito in un tempo $O(n^2)$. Analogamente, un algoritmo il cui tempo d'esecuzione è al massimo pari a $5n + 20 \log n + 4$ sarà un algoritmo *tempo-lineare*.

Omega-grande (Ω)

Così come la notazione O-grande consente di dire se una funzione è asintoticamente "minore di o uguale a" un'altra funzione, la notazione che segue consente di dire se una funzione cresce a una velocità che è "maggior di o uguale a" quella di un'altra funzione.

Siano $f(n)$ e $g(n)$ funzioni che mettono in corrispondenza numeri interi positivi con numeri reali positivi. Diciamo che $f(n) \in \Omega(g(n))$ (che si legge " $f(n)$ è Omega-grande di $g(n)$ ") se $g(n) \in O(f(n))$, cioè se esiste una costante reale $c > 0$ e una costante intera $n_0 \geq 1$ tali che:

$$f(n) \geq cg(n), \text{ per } n \geq n_0.$$

Questa definizione consente di dire che una funzione è asintoticamente "maggiore di o uguale a" (cioè "non minore di") un'altra funzione, a meno di un fattore costante.

Esempio 4.14: $3n \log n - 2n \in \Omega(n \log n)$.

Dimostrazione: $3n \log n - 2n = n \log n + 2n(\log n - 1) \geq n \log n$, per $n \geq 2$. Quindi, in questo caso basta prendere $c = 1$ e $n_0 = 2$. ■

Theta-grande (Θ)

Infine, esiste una notazione che consente di dire se due funzioni crescono con la stessa velocità, a meno di un fattore costante. Diciamo che $f(n) \in \Theta(g(n))$ (che si legge " $f(n)$ è Theta-grande di $g(n)$ ") se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$, cioè se esistono due costanti reali $c' > 0$ e $c'' > 0$ e una costante intera $n_0 \geq 1$ tali che:

$$c' g(n) \leq f(n) \leq c'' g(n), \text{ per } n \geq n_0.$$

Esempio 4.15: $3n \log n + 4n + 5 \log n \in \Theta(n \log n)$.

Dimostrazione: $3n \log n \leq 3n \log n + 4n + 5 \log n \leq (3 + 4 + 5) n \log n$, per $n \geq 2$. ■

4.3.2 Analisi comparativa

La notazione O-grande è ampiamente utilizzata per caratterizzare i tempi di esecuzione e lo spazio occupato in funzione di un parametro n , che viene definito come una misura adeguata della "dimensione" del problema. Supponiamo che siano disponibili due algoritmi che risolvono uno stesso problema: un algoritmo A che ha un tempo d'esecuzione $O(n)$, e un algoritmo B che ha un tempo d'esecuzione $O(n^2)$. Quale algoritmo è migliore? Sappiamo che $n = O(n^2)$, per cui l'algoritmo A è **asintoticamente migliore** dell'algoritmo B , anche se, per valori di n piccoli, B potrebbe avere un tempo d'esecuzione inferiore di quello di A .

Possiamo usare la notazione O-grande per ordinare classi di funzioni in base alla loro velocità di crescita asintotica. Le nostre sette funzioni sono ordinate per velocità crescente in questo modo, con $f(n)$ che è $O(g(n))$ se la funzione $f(n)$ precede la funzione $g(n)$ nell'ordinamento:

$$1, \quad \log n, \quad n, \quad n \log n, \quad n^2, \quad n^3, \quad 2^n.$$

Nella Tabella 4.3 possiamo vedere le velocità di crescita delle sette funzioni (che si può dedurre anche dalla Figura 4.4 del Paragrafo 4.2.1).

Tabella 4.3: Alcuni valori assunti dalle funzioni fondamentali usate nell'analisi di algoritmi

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4294967296
64	6	64	384	4096	262144	1.84×10^{19}
128	7	128	896	16384	2097152	3.40×10^{38}
256	8	256	2048	65536	16777216	1.15×10^{77}
512	9	512	4608	262144	134217728	1.34×10^{154}

Nella Tabella 4.4 rendiamo evidente l'importanza del punto di vista asintotico. Questa tabella analizza la dimensione massima consentita per un esemplare del problema che viene elaborato da un algoritmo in un secondo, in un minuto e in un'ora, e mostra quanto sia determinante un buon progetto dell'algoritmo, dal momento che un algoritmo asintoticamente lento viene battuto, per problemi sufficientemente grandi, da algoritmi asintoticamente più veloci, anche se il loro fattore costante è peggiore.

Tabella 4.4: Dimensione massima del problema che può essere risolto in un secondo, un minuto o un'ora, per diverse funzioni del tempo d'esecuzione, misurato in microsecondi

Tempo d'esecuzione (μs)	Dimensione massima del problema (n)		
	1 secondo	1 minuto	1 ora
$400n$	2500	150000	9000000
$2n^2$	707	5477	42426
2^n	19	25	31

L'importanza di un buon progetto di algoritmo va, però, ben oltre ciò che può effettivamente essere risolto con un determinato calcolatore. Come si può vedere nella Tabella 4.5, anche se riuscissimo a ottenere un computer dotato di hardware decisamente più veloce, non saremmo comunque in grado di vincere lo svantaggio derivante da un algoritmo asintoticamente lento. Questa tabella mostra i nuovi valori della dimensione massima del problema risolvibile in un tempo massimo prefissato, nell'ipotesi che gli algoritmi, di cui sono indicati i tempi di esecuzione, vengano ora eseguiti su un computer 256 volte più veloce di quello precedente.

Tabella 4.5: Aumento della dimensione massima del problema che può essere risolto in una quantità di tempo prefissata, usando un computer 256 volte più veloce di quello precedente. Ogni dato è funzione di m , la dimensione massima precedente.

Tempo d'esecuzione	Nuova dimensione massima del problema
$400n$	$256m$
$2n^2$	$16m$
2^n	$m + 8$

Qualche precauzione

È giunto il momento di spendere alcune parole di cautela in relazione alla notazione asintotica. Innanzitutto, va detto che l'uso della notazione O -grande e delle notazioni a essa correlate può a volte essere fuorviante, qualora i fattori costanti "nascosti" da tali notazioni siano molto grandi. Ad esempio, sebbene sia vero che la funzione $10^{100}n$ è $O(n)$, se questa rappresenta il tempo d'esecuzione di un algoritmo che viene confrontato con un altro il cui tempo è $10n \log n$, dovremmo preferire l'algoritmo che è $O(n \log n)$, nonostante l'algoritmo tempo-lineare sia asintoticamente più veloce. Questa decisione è dovuta al fatto che il fattore costante, 10^{100} (un numero che viene anche detto "un googol"), è ritenuto da molti astrofisici un limite superiore al numero di atomi presenti nell'universo osservabile, per cui è altamente improbabile che si presenti un problema, nel mondo reale, la cui dimensione sia così grande.

L'osservazione precedente fa emergere un problema: quand'è che un algoritmo è "veloce"? Parlando in generale, qualunque algoritmo che abbia un tempo d'esecuzione $O(n \log n)$ (con un fattore costante ragionevole) dovrebbe essere considerato efficiente. In alcune situazioni, cioè quando n è piccolo, anche un algoritmo tempo-quadratico potrebbe essere abbastanza veloce. Ma un algoritmo il cui tempo d'esecuzione sia una funzione esponenziale, ad esempio $O(2^n)$, non dovrebbe mai essere considerato efficiente.

Tempi d'esecuzione esponenziali

Per vedere quanto cresca velocemente la funzione 2^n , consideriamo la famosa leggenda sull'inventore del gioco degli scacchi. Egli chiese soltanto che il suo re gli pagasse un chicco di riso per la prima casella della scacchiera, 2 chicchi per la seconda, 4 chicchi per la terza, 8 chicchi per la quarta, e così via. Il numero di chicchi per la 64-esima casella della scacchiera sarebbe $2^{63} = 9223372036854775808$, cioè circa nove miliardi di miliardi!

Se dovessimo tracciare una linea di separazione tra algoritmi efficienti e inefficienti, sarebbe quindi naturale distinguere gli algoritmi che vengono eseguiti in un tempo polinomiale da quelli che richiedono un tempo esponenziale. I primi sono caratterizzati da

un tempo d'esecuzione che è $O(n)$ per una qualche costante $c \geq 1$, mentre i secondi sono $O(b^n)$ per una qualche costante $b > 1$. Come molte nozioni di cui abbiamo discusso in questo paragrafo, anche questa va presa "cum grano salis", cioè con qualche precauzione, perché probabilmente un algoritmo che viene eseguito in un tempo $O(n^{100})$, pur essendo polinomiale, non dovrebbe essere considerato "efficiente". In ogni caso, la distinzione tra algoritmi tempo-polinomiali e algoritmi tempo-esponenziali viene considerata un metro di valutazione della trattabilità piuttosto solido.

4.3.3 Esempi di analisi di algoritmi

Ora che abbiamo a disposizione la notazione O-grande per l'analisi degli algoritmi, vediamo alcuni esempi, nei quali caratterizzeremo il tempo d'esecuzione di alcuni semplici algoritmi usando tale notazione. Inoltre, manterremo la promessa di usare ciascuna delle sette funzioni di cui abbiamo parlato per caratterizzare il tempo d'esecuzione di uno degli esempi di algoritmo.

Operazioni tempo-costanti

Tutte le operazioni elementari che abbiamo descritto nel Paragrafo 4.1.1 vengono eseguite in un tempo costante e, in modo più formale, diciamo che vengono eseguite in un tempo $O(1)$. Vogliamo, in particolare, porre l'accento su alcune importanti operazioni tempo-costanti che riguardano gli array. Ipotizziamo che la variabile A sia un array di n elementi. L'espressione $A.length$, in Java, viene valutata in un tempo costante, perché la rappresentazione interna degli array contiene una variabile esplicita che memorizza la lunghezza dell'array. Un altro comportamento chiave degli array è il fatto che si possa accedere in un tempo costante a qualunque suo singolo elemento, $A[j]$, dato un indice valido, j . Questo avviene perché un array usa un unico blocco di memoria contenente tutti gli elementi, uno consecutivo all'altro nell'ordine specificato dai loro indici: quindi, per trovare l'elemento j -esimo non è necessario scandire l'array un elemento per volta, ma, dopo aver determinato che l'indice sia valido, lo si può usare come valore dello spostamento a partire dall'inizio dell'array, determinando l'indirizzo in memoria corretto per l'elemento cercato. Anche in questo caso, diciamo che l'espressione $A[j]$ viene valutata, per un array, in un tempo $O(1)$.

Trovare l'elemento massimo in un array

Come classico esempio di algoritmo il cui tempo d'esecuzione cresce proporzionalmente a n , consideriamo l'obiettivo di individuare l'elemento massimo in un array. Una strategia tipica prevede di scandire tutti gli elementi dell'array, uno dopo l'altro, tenendo continuamente traccia in una variabile dell'elemento massimo visto fino a quel momento. Il Codice 4.3 presenta il metodo `arrayMax` che implementa proprio questa strategia.

Codice 4.3: Metodo che restituisce il valore massimo di un array.

```

1  /** Restituisce il valore massimo in un array di numeri non vuoto. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;
4      double currentMax = data[0]; // ipotizza che il primo sia il massimo (per ora)
5      for (int j=1; j < n; j++) // esamina tutti gli altri valori
6          if (data[j] > currentMax) // se data[j] è il massimo fin qui...

```

```

7   currentMax = data[j];    // memorizzalo come massimo attuale
8   return currentMax;
9 }
```

Usando la notazione O-grande, possiamo scrivere il seguente enunciato matematico, che descrive con precisione il tempo impiegato dall'algoritmo implementato da `arrayMax` per l'esecuzione su *qualsiasi* calcolatore.

Proposizione 4.16: *L'algoritmo `arrayMax` che individua l'elemento massimo in un array di n numeri viene eseguito in un tempo $O(n)$.*

Dimostrazione: La fase di inizializzazione, alle righe 3 e 4, richiede soltanto un numero costante di operazioni elementari, così come l'enunciato `return` alla riga 8. Ogni iterazione del ciclo richiede, di nuovo, un numero costante di operazioni elementari, e il ciclo viene eseguito $n - 1$ volte. Quindi, possiamo dire che il numero totale di operazioni elementari è $c' \cdot (n - 1) + c''$, dove c' e c'' sono costanti opportune che tengono conto, rispettivamente, del lavoro svolto all'interno e all'esterno del corpo del ciclo. Dato che ogni operazione elementare viene eseguita in un tempo costante, il tempo d'esecuzione dell'algoritmo `arrayMax` su un array di dimensione n è, al massimo, $c' \cdot (n - 1) + c'' = c' \cdot n + (c'' - c') \leq c' \cdot n$, se ipotizziamo, senza per questo perdere generalità, che sia $c'' \leq c'$. Possiamo, quindi, concludere che il tempo d'esecuzione dell'algoritmo `arrayMax` è $O(n)$. ■

Un'ulteriore analisi dell'algoritmo che trova l'elemento massimo

Una domanda a cui può essere interessante rispondere in relazione all'algoritmo `arrayMax` è il conteggio dei possibili aggiornamenti della variabile che memorizza il massimo valore visto fino a quel momento. Nel caso peggiore, cioè quando i dati sono disposti in ordine crescente all'interno dell'array, il valore massimo viene riassegnato $n - 1$ volte. Ma cosa succede se i dati in ingresso sono disposti in ordine casuale, con tutti i possibili ordinamenti equamente probabili? Qual è il numero atteso di aggiornamenti in tal caso? Per rispondere a questa domanda, osserviamo che aggiorniamo il valore di tale variabile durante un'iterazione del ciclo soltanto se l'elemento in esame è maggiore di tutti gli elementi che lo precedono. Se la sequenza è in ordine casuale, la probabilità che il j -esimo elemento sia il massimo tra i primi j elementi è $1/j$ (ipotizzando che gli elementi siano tutti diversi). Quindi, il numero atteso di aggiornamenti della variabile che memorizza il valore massimo (compresa la sua inizializzazione) è $H_n = \sum_{j=1}^n 1/j$, un valore che è noto con il nome di *n-esimo numero armonico*. Si può dimostrare che H_n è $O(\log n)$, quindi il numero atteso di aggiornamenti della variabile che memorizza il valore massimo durante l'esecuzione di `arrayMax` su una sequenza in ordine casuale è $O(\log n)$.

Costruire stringhe lunghe

In questo esempio, rivediamo l'analisi sperimentale che avevamo condotto nel Paragrafo 4.1, nella quale avevamo esaminato due diverse implementazioni della costruzione di una stringa lunga (nel Codice 4.2). Il nostro primo algoritmo era basato sull'uso ripetuto dell'operatore di concatenazione tra stringhe (per comodità, ripetiamo il metodo nel Codice 4.4).

Codice 4.4: Costruzione di una stringa mediante concatenazione ripetuta.

```

1  /** Usa la concatenazione per costruire una stringa con n copie del carattere c. */
2  public static String repeat1(char c, int n) {
3      String answer = "";
4      for (int j=0; j < n; j++)
5          answer += c;
6      return answer;
7  }

```

L'aspetto più importante di questa implementazione è che, in Java, le stringhe sono oggetti *immutable*: una volta creato, un esemplare di stringa non può essere modificato. L'enunciato `answer += c` è un'abbreviazione di `answer = (answer + c)`: questo secondo enunciato non aggiunge un nuovo carattere (`c`) all'esemplare di `String` esistente, ma genera un nuovo esemplare di `String` con la sequenza di caratteri richiesta, poi ne assegna il riferimento alla variabile `answer`, che, in seguito, farà riferimento a tale nuova stringa.

In termini di efficienza, il problema di questa implementazione è che la creazione di una nuova stringa come risultato della concatenazione richiede un tempo che è proporzionale alla lunghezza della stringa prodotta. Durante la prima iterazione del ciclo il risultato ha lunghezza 1, durante la seconda ha lunghezza 2, e così via, finché non si raggiunge la lunghezza dell'ultima stringa, che è n . Di conseguenza, il tempo totale richiesto da questo algoritmo è proporzionale a $1 + 2 + \dots + n$, che riconosciamo come la ben nota sommatoria della Proposizione 4.3, che è $O(n^2)$. Quindi, il tempo totale richiesto dall'esecuzione dell'algoritmo `repeat1` è $O(n^2)$.

Possiamo vedere come questa analisi teorica si rifletta nei risultati sperimentali. Il tempo d'esecuzione di un algoritmo quadratico dovrebbe, in teoria, quadruplicarsi se la dimensione del problema raddoppia, perché $(2n)^2 = 4 \cdot n^2$ (diciamo "in teoria" perché non teniamo conto dei termini di grado inferiore a quello massimo, che sono nascosti dalla notazione asintotica). Osservando la Tabella 4.1, si nota effettivamente una tale quadruplicazione del tempo d'esecuzione di `repeat1`.

In quella stessa tabella, invece, i tempi d'esecuzione riportati per l'algoritmo `repeat2`, che usa la classe `StringBuilder` della libreria di Java, dimostrano la tendenza a *raddoppiare* approssimativamente ogni volta che la dimensione del problema raddoppia. La classe `StringBuilder` si basa su una tecnica avanzata, con un tempo d'esecuzione $O(n)$ nel caso peggiore per la costruzione di una stringa di lunghezza n : studieremo questa tecnica nel Paragrafo 7.2.1.

Intersezione vuota di tre insiemi

Supponiamo che siano dati tre insiemi, A , B e C , memorizzati in tre diversi array di numeri interi. Faremo l'ipotesi che nessuno degli insiemi contenga valori duplicati al proprio interno, ma alcuni numeri possono appartenere a due o tre degli insiemi. Il problema dell'*intersezione vuota di tre insiemi* (*three-way set disjointness*) consiste nel determinare se l'intersezione dei tre insiemi è vuota, cioè se non esiste alcun elemento x tale che $x \in A$, $x \in B$ e $x \in C$. Il Codice 4.5 riporta un semplice metodo che, in Java, determina tale proprietà.

Codice 4.5: L'algoritmo `disjoint1` per verificare se tre insiemi hanno intersezione vuota.

```

1  /** Restituisce true se e solo se non esiste un elemento comune ai tre array. */
2  public static boolean disjoint1(int[] groupA, int[] groupB, int[] groupC) {
3      for (int a : groupA)
4          for (int b : groupB)

```

```

5   for (int c : groupC)
6     if ((a == b) && (b == c))
7       return false; // abbiamo trovato un valore comune a tutti gli array
8     return true;      // se arriviamo qui, gli insiemi sono disgiunti
9   }

```

Questo semplice algoritmo scandisce tutte le possibili triplette di valori composte da un elemento preso da ciascun insieme, cercandone una che contenga valori uguali. Se ciascuno degli insiemi ha dimensione n , il tempo d'esecuzione di questo metodo nel caso peggiore è $O(n^3)$.

Possiamo migliorarne le prestazioni asintotiche con una semplice osservazione: all'interno del corpo del ciclo che scandisce B , se gli elementi a e b selezionati non sono uguali tra loro, scandire tutti gli elementi di C cercando una tripletta di valori identici è una perdita di tempo. Il Codice 4.6 presenta una soluzione migliore di questo problema, sfruttando quest'ultima osservazione.

Codice 4.6: L'algoritmo `disjoint2` per verificare se tre insiemi hanno intersezione vuota.

```

1  /** Restituisce true se e solo se non esiste un elemento comune ai tre array. */
2  public static boolean disjoint2(int[] groupA, int[] groupB, int[] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        if (a == b)          // analizza C soltanto se a e b sono uguali
6          for (int c : groupC)
7            if (a == c)        // e quindi sarà anche b == c
8              return false; // abbiamo trovato un valore comune a tutti gli array
9    return true;           // se arriviamo qui, gli insiemi sono disgiunti
10   }

```

Nella versione migliorata, `disjoint2`, non si tratta semplicemente di risparmiare tempo nei casi fortunati: affermiamo che il tempo d'esecuzione *nel caso peggiore* è $O(n^2)$. Il numero di coppie (a, b) da considerare è quadratico, ma, se A e B sono entrambi insiemi di elementi distinti, il numero di tali coppie che abbiano a uguale a b può, al massimo, essere una quantità $O(n)$. Quindi, il ciclo più interno, che scandisce gli elementi di C , al massimo viene fatto iniziare n volte.

Per calcolare il tempo d'esecuzione totale, esaminiamo il tempo dedicato all'esecuzione di ciascuna linea di codice. La gestione del ciclo `for` che opera su A richiede un tempo $O(n)$. La gestione del ciclo `for` che opera su B richiede un tempo totale $O(n^2)$, perché tale ciclo viene eseguito n volte. Il confronto $a == b$ viene eseguito $O(n^2)$ volte. Il resto del tempo impiegato dipende da quante coppie (a, b) di elementi uguali esistono. Come abbiamo osservato, il numero massimo di queste coppie è n , quindi la gestione del ciclo che opera su C e degli enunciati all'interno del corpo di tale ciclo impiegano al massimo un tempo $O(n^2)$. Applicando la Proposizione 4.8, il tempo d'esecuzione totale è $O(n^2)$.

Unicità degli elementi

Un problema strettamente correlato a quello dell'intersezione vuota di tre insiemi è l'*unicità degli elementi*. Nel primo erano dati tre insiemi e ipotizzavamo che all'interno di ciascuno di essi non vi fossero elementi duplicati. Nel problema dell'unicità degli elementi, viene fornito un array con n elementi e si chiede di determinare se tutti i suoi elementi sono tra loro distinti.

La nostra prima soluzione di questo problema' usa un algoritmo iterativo abbastanza banale. Il metodo `unique1`, riportato nel Codice 4.7, risolve il problema dell'unicità degli elementi eseguendo una scansione di tutte le coppie (j, k) di indici distinti, con $j < k$, verificando se una di tali coppie si riferisce a elementi uguali tra loro. Per farlo usa due cicli `for` annidati, in modo che la prima iterazione del ciclo esterno provochi l'esecuzione di $n - 1$ iterazioni del ciclo interno, la seconda iterazione del ciclo esterno provochi l'esecuzione di $n - 2$ iterazioni del ciclo interno, e così via. Quindi, il tempo d'esecuzione di questo metodo nel caso peggiore è proporzionale a $(n - 1) + (n - 2) + \dots + 2 + 1$, che riconosciamo essere la ben nota sommatoria della Proposizione 4.3, il cui valore è $O(n^2)$.

Codice 4.7: L'algoritmo `unique1` per verificare l'unicità degli elementi di un array.

```

1  /** Restituisce true se e solo se non esistono elementi duplicati nell'array. */
2  public static boolean unique1(int[] data) {
3      int n = data.length;
4      for (int j=0; j < n-1; j++)
5          for (int k=j+1; k < n; k++)
6              if (data[j] == data[k])
7                  return false; // trovata una coppia di elementi duplicati
8      return true;      // se arriviamo qui, gli elementi sono tutti diversi
9  }

```

L'ordinamento come strumento di soluzione dei problemi

Un algoritmo migliore per risolvere il problema dell'unicità degli elementi si basa sull'uso dell'ordinamento come strumento per risolvere problemi. In questo caso, ordinando l'array di elementi, abbiamo la garanzia che eventuali elementi duplicati si verranno a trovare in posizioni tra loro adiacenti. Quindi, per determinare se ci sono elementi duplicati, ci basta eseguire un'unica scansione dell'array ordinato, cercando elementi *consecutivi* duplicati.

Nel Codice 4.8 vediamo un'implementazione in Java di questo algoritmo (la classe `java.util.Arrays` è stata discussa nel Paragrafo 3.1.3).

Codice 4.8: L'algoritmo `unique2` per verificare l'unicità degli elementi di un array.

```

1  /** Restituisce true se e solo se non esistono elementi duplicati nell'array. */
2  public static boolean unique2(int[] data) {
3      int n = data.length;
4      int[] temp = Arrays.copyOf(data, n); // fa una copia dei dati
5      Arrays.sort(temp);                // e ordina la copia
6      for (int j=0; j < n-1; j++)
7          if (temp[j] == temp[j+1]) // controlla una coppia di elementi consecutivi
8              return false; // trovata una coppia di elementi duplicati
9      return true;      // se arriviamo qui, gli elementi sono tutti diversi
10 }

```

Gli algoritmi di ordinamento saranno argomento del Capitolo 12. I migliori algoritmi di ordinamento (tra i quali quelli usati dal metodo `Arrays.sort` della libreria di Java) garantiscono un tempo d'esecuzione $O(n \log n)$ nel caso peggiore. Dopo aver ordinato i dati, il ciclo che segue viene eseguito in un tempo $O(n)$, per cui l'intero algoritmo `unique2` richiede un tempo $O(n \log n)$. L'Esercizio C-4.35 studia l'uso dell'ordinamento per risolvere il problema dell'intersezione vuota tra tre insiemi in un tempo $O(n \log n)$.

Medie dei precedenti

Il prossimo problema che esaminiamo è il calcolo delle cosiddette *medie dei precedenti* o *dei prefissi* (*prefix averages*) di una sequenza di numeri. Data una sequenza x costituita da n numeri, vogliamo calcolare una sequenza a tale che a_j sia la media degli elementi x_0, \dots, x_j , con $j = 0, \dots, n - 1$, cioè:

$$a_j = \frac{\sum_{i=0}^j x_i}{j+1}$$

Le medie dei precedenti hanno molte applicazioni in economia e statistica. Ad esempio, dati i rendimenti anno per anno di un fondo comune di investimento, ordinati dal presente al passato, un investitore vorrà tipicamente conoscere il rendimento medio annuo del fondo nell'ultimo anno, negli ultimi tre anni, negli ultimi cinque anni, e così via. Analogamente, dato un flusso di dati relativi all'utilizzo giornaliero di pagine web, il gestore di un sito potrebbe voler calcolare l'andamento dell'utilizzo medio su vari periodi di tempo. Presentiamo, quindi, due diverse soluzioni per il calcolo delle medie dei precedenti, con tempi d'esecuzione significativamente diversi.

Un algoritmo tempo-quadratico

Il nostro primo algoritmo per il calcolo delle medie dei precedenti, che chiamiamo `prefixAverage1`, è riportato nel Codice 4.9 e calcola ciascun valore a_j indipendentemente dagli altri, usando un ciclo interno che calcola la somma parziale necessaria.

Codice 4.9: L'algoritmo `prefixAverage1`.

```

1  /** Restituisce un array con a[j] = alla media di x[0],...,x[j], per ogni j. */
2  public static double[] prefixAverage1(double[] x) {
3      int n = x.length;
4      double[] a = new double[n]; // viene creato e riempito di zeri
5      for (int j=0; j < n; j++) {
6          double total = 0; // inizia il calcolo di x[0] + ... + x[j]
7          for (int i=0; i <= j; i++)
8              total += x[i];
9          a[j] = total / (j+1); // memorizza la media calcolata
10     }
11  return a;
12 }
```

Analizziamo l'algoritmo `prefixAverage1`.

- L'inizializzazione $n = x.length$ alla riga 3 e la restituzione conclusiva del riferimento all'array a alla riga 11 vengono entrambe eseguite in un tempo $O(1)$.
- La creazione e inizializzazione del nuovo array, a , alla riga 4 viene eseguita in un tempo $O(n)$, perché usa un numero costante di operazioni elementari per ciascun elemento.
- Ci sono due cicli `for` annidati, controllati, rispettivamente, dai contatori j e i . Il corpo del ciclo esterno, controllato dal contatore j , viene eseguito n volte, per $j = 0, \dots, n - 1$, quindi gli enunciati $total = 0$ e $a[j] = total / (j+1)$ vengono eseguiti n volte ciascuno. Questo implica che questi due enunciati, a cui si aggiunge la gestione del contatore j

nel ciclo, contribuiscono con un numero di operazioni elementari proporzionale a n , cioè un tempo $O(n)$.

- Il corpo del ciclo interno, controllato dal contatore i , viene eseguito $j + 1$ volte, in funzione del valore di j , il contatore del ciclo esterno. Di conseguenza, l'enunciato $\text{total} += x[i]$, nel ciclo interno, viene eseguito $1 + 2 + 3 + \dots + n$ volte. Ricordando la Proposizione 4.3, sappiamo che $1 + 2 + 3 + \dots + n = n(n + 1)/2$, per cui l'enunciato che costituisce il corpo del ciclo interno contribuisce con un tempo $O(n^2)$ e considerazioni analoghe si possono fare per le operazioni elementari associate alla gestione del contatore i , che richiedono di nuovo un tempo $O(n^2)$.

Il tempo d'esecuzione del metodo `prefixAverage1` è dato dalla somma di questi termini. Il primo termine è $O(1)$, il secondo e il terzo sono $O(n)$, mentre il quarto è $O(n^2)$. La semplice applicazione della Proposizione 4.8 dice che il tempo d'esecuzione di `prefixAverage1` è $O(n^2)$.

Un algoritmo tempo-lineare

Nel calcolo delle medie dei precedenti si trova un valore intermedio, la *somma dei precedenti*, $x_0 + x_1 + \dots + x_j$, che nella nostra prima implementazione era memorizzata nella variabile `total`, consentendoci così di calcolare la media dei precedenti con la formula $a[j] = total / (j+1)$. Nel nostro primo algoritmo, la somma dei precedenti viene ricalcolata completamente, a partire dall'inizio, per ogni valore di j : questo richiede un tempo $O(j)$ per ogni valore di j , dando luogo al comportamento quadratico del tempo d'esecuzione.

Per una maggior efficienza, possiamo gestire la somma dei precedenti in modo dinamico, calcolando $x_0 + x_1 + \dots + x_j$ come `total + xj`, dove il valore di `total` è uguale alla somma $x_0 + x_1 + \dots + x_{j-1}$ calcolata durante la precedente iterazione del ciclo controllato da j . Il Codice 4.10 presenta la nuova implementazione, `prefixAverage2`, che usa questo approccio.

Codice 4.10: L'algoritmo `prefixAverage2`.

```

1  /** Restituisce un array con a[j] = alla media di x[0],...,x[j], per ogni j. */
2  public static double[] prefixAverage2(double[] x) {
3      int n = x.length;
4      double[] a = new double[n]; // viene creato e riempito di zeri
5      double total = 0; // calcola la somma dei precedenti come x[0]+x[1]+...
6      for (int j=0; j < n; j++) {
7          total += x[j]; // aggiorna la somma dei precedenti aggiungendo x[j]
8          a[j] = total / (j+1); // memorizza la media calcolata
9      }
10     return a;
11 }
```

Ecco l'analisi del tempo d'esecuzione dell'algoritmo `prefixAverage2`.

- L'inizializzazione delle variabili `n` e `total` richiede un tempo $O(1)$.
- La creazione e inizializzazione dell'array `a` richiede un tempo $O(n)$.
- C'è un unico ciclo `for`, controllato dal contatore j , la cui gestione dà un contributo $O(n)$ al tempo totale.
- Il corpo del ciclo viene eseguito n volte, per $j = 0, \dots, n - 1$, quindi gli enunciati `total += x[j]` e `a[j] = total / (j+1)` vengono eseguiti n volte ciascuno. Dato che ognuno di

questi enunciati viene eseguito in un tempo $O(1)$, il loro contributo complessivo al tempo d'esecuzione è $O(n)$.

- La restituzione finale del riferimento all'array a è $O(1)$.

Il tempo d'esecuzione dell'algoritmo `prefixAverage2` è, infine, dato dalla somma dei cinque termini appena elencati. Il primo e l'ultimo sono $O(1)$ e gli altri tre sono $O(n)$. Applicando nuovamente la Proposizione 4.8, il tempo d'esecuzione di `prefixAverage2` è $O(n)$, decisamente migliore del tempo quadratico richiesto dall'algoritmo `prefixAverage1`.

4.4 Semplici tecniche di dimostrazione

A volte vorremo enunciare affermazioni relative a un algoritmo, come il fatto che sia corretto o che venga eseguito velocemente. Per farlo in modo rigoroso, dobbiamo usare il linguaggio tipico della matematica e, per sostenere tali affermazioni, dobbiamo darne una giustificazione o *dimostrazione*. Fortunatamente, ci sono parecchi modi per farlo.

4.4.1 Dimostrare con un esempio

Alcune affermazioni hanno genericamente questa forma: "Esiste un elemento x in un insieme S che gode della proprietà P ". Per dimostrare un'affermazione di questo tipo, dobbiamo solamente individuare un particolare elemento x all'interno di S che goda effettivamente della proprietà P . Analogamente, alcune altre affermazioni che si ritengono false hanno la forma: "Ogni elemento x in un insieme S gode della proprietà P ". Per dimostrare che una tale affermazione è falsa, è sufficiente individuare un elemento x appartenente a S che *non* gode della proprietà P : un tale elemento viene chiamato *controesempio*.

Esempio 4.17: Il Professor Amongus afferma che qualsiasi numero esprimibile come $2^i - 1$, con i numero intero maggiore di 1, è un numero primo. Il Professor Amongus ha torto.

Dimostrazione: Per dimostrare che il Professor Amongus ha torto è sufficiente trovare un controesempio. Fortunatamente, non occorre andare tanto lontano, perché $2^4 - 1 = 15 = 3 \cdot 5$. ■

4.4.2 Dimostrare per contrapposizione o contraddizione

Un altro insieme di tecniche di dimostrazione prevede di utilizzare la negazione logica. I due metodi principali usano la *contrapposizione* e la *contraddizione*. Per dimostrare l'affermazione "se p è vero, allora q è vero", possiamo invece cercare di dimostrare che "se q non è vero, allora p non è vero". Dal punto di vista logico, questi due enunciati si equivalgono, ma può essere più semplice ragionare sul secondo, che è la *contrapposizione* del primo.

Esempio 4.18: Siano a e b numeri interi. Se ab è un numero pari, allora a è pari oppure b è pari.

Dimostrazione: Per dimostrare questa affermazione, consideriamo la sua contrapposizione: “se a è dispari e b è dispari, allora ab è dispari”. Supponiamo, quindi, che esistano numeri interi j e k tali che $a = 2j + 1$ e $b = 2k + 1$. Allora $ab = 4jk + 2j + 2k + 1 = 2(2jk + j + k) + 1$, quindi ab è dispari. ■

L'esempio precedente, oltre a illustrare la tecnica di dimostrazione mediante contrapposizione, contiene anche un'applicazione della *legge di de Morgan*, che è di ausilio nella gestione della negazione logica, perché afferma che la negazione di un enunciato avente la forma “ p oppure q ” è “non p e non q ”. Analogamente, afferma’ che la negazione di un enunciato del tipo “ p e q ” è “non p oppure non q ”.

Contraddizione

Un'altra tecnica di dimostrazione mediante negazione logica è la dimostrazione per *contraddizione*, che spesso richiede l'utilizzo della legge di de Morgan. Usando la tecnica della dimostrazione per contraddizione, determiniamo che l'enunciato q è vero supponendo prima che sia falso, per poi giungere alla conclusione che tale ipotesi porta a una contraddizione (come $2 \neq 2$ oppure $1 > 3$). Giungendo a una tale contraddizione, dimostriamo che non esiste alcuna situazione coerente in cui q sia falso, per cui q deve essere vero. Ovviamente, per poter trarre una simile conclusione, dobbiamo essere certi che quanto affermiamo sia coerente con il fatto che q sia falso.

Esempio 4.19: Siano a e b numeri interi. Se ab è un numero dispari, allora a è dispari e b è dispari.

Dimostrazione: Sia ab un numero dispari: vogliamo dimostrare che a è dispari e b è dispari. Nella speranza di poter giungere a una contraddizione, ipotizziamo il contrario della tesi, cioè supponiamo che a sia pari oppure b sia pari. In effetti, senza perdere in generalità, possiamo ipotizzare che a sia pari, dal momento che il problema è simmetrico. Di conseguenza, esisterà un numero intero j tale che $a = 2j$, per cui $ab = (2j)b = 2(jb)$: questo implica che ab sia un numero pari. Ma questa è una contraddizione: ab non può essere contemporaneamente dispari e pari, quindi a è dispari e b è dispari. ■

4.4.3 Dimostrare per induzione o mediante invariante di ciclo

La maggior parte delle affermazioni che facciamo sul tempo d'esecuzione o sullo spazio occupato da un algoritmo riguarda un parametro intero n (che solitamente rappresenta un valore della “dimensione” del problema, un concetto che riteniamo spesso intuitivo). Inoltre, molte di queste affermazioni sono equivalenti a enunciati del tipo “ $q(n)$ è vero per ogni valore $n \geq 1$ ”. Trattandosi di un'affermazione che riguarda un insieme infinito di numeri, non possiamo dimostrarla in modo diretto.

Induzione

Spesso, però, siamo in grado di dimostrare che enunciati come quelli appena menzionati sono veri usando la tecnica dell'*induzione*. Questa tecnica mira a dimostrare che, per qualsiasi valore di $n \geq 1$, esiste una sequenza finita di implicazioni che partono da qualcosa che si sa essere vero e terminano con la dimostrazione che $q(n)$ è un enunciato vero. Nello spe-

cifico, si inizia una dimostrazione per induzione dimostrando che $q(n)$ è vero per $n = 1$ (e magari anche per altri valori $n = 2, 3, \dots, k$, con k costante). Poi, si dimostra che il "passo" induttivo è vero per $n > k$, cioè si dimostra che "se $q(j)$ è vero per qualunque $j < n$, allora $q(n)$ è vero". La combinazione di queste due parti completa la dimostrazione per induzione.

Proposizione 4.20: Consideriamo la funzione di Fibonacci $F(n)$, definita in modo che $F(1) = 1$, $F(2) = 2$ e $F(n) = F(n - 2) + F(n - 1)$ per $n > 2$ (si veda il Paragrafo 2.2.3). Affermiamo che $F(n) < 2^n$.

Dimostrazione: Dimostreremo la correttezza della nostra affermazione mediante induzione.

Caso base: $n \leq 2$. $F(1) = 1 < 2 = 2^1$ e $F(2) = 2 < 4 = 2^2$.

Passo induttivo: $n > 2$. Supponiamo che l'affermazione sia vera per ogni $j < n$. Dato che tanto $n - 2$ quanto $n - 1$ sono minori di n , possiamo ritenere valida l'*ipotesi induttiva*, che implica:

$$F(n) = F(n - 2) + F(n - 1) < 2^{n-2} + 2^{n-1}.$$

Essendo:

$$2^{n-2} + 2^{n-1} < 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n,$$

otteniamo che $F(n) < 2^n$, dimostrando così il passo induttivo. ■

Facciamo un altro esempio, questa volta per un'affermazione che abbiamo già dimostrato in precedenza.

Proposizione 4.21: (già vista come Proposizione 4.3)

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Dimostrazione: Dimostreremo questa uguaglianza mediante induzione.

Caso base: $n = 1$. Banale, perché $1 = n(n+1)/2$ se $n = 1$.

Passo induttivo: $n \geq 2$. Supponiamo che l'ipotesi induttiva sia vera per ogni $j < n$. Quindi, per $j = n - 1$ abbiamo:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}$$

da cui otteniamo:

$$\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2},$$

dimostrando così il passo induttivo. ■

A volte potremmo pensare che il compito di dimostrare che un'affermazione è vera per qualsiasi $n \geq 1$ sia troppo oneroso: dovremmo, in ogni caso, tenere a mente la potenza della tecnica induttiva. Essa dimostra che, per qualsiasi valore di n , esiste una sequenza finita di implicazioni che inizia con un'affermazione vera e termina dimostrando una verità relativa al valore n . In breve, la tecnica induttiva è uno schema che consente di costruire una sequenza di implicazioni che si sostengono in modo diretto, una di seguito all'altra.

Invarianti di ciclo

L'ultima tecnica di dimostrazione di cui parliamo in questo paragrafo è l'*invariante di ciclo* (o, per meglio dire, la *condizione invariante in un ciclo*). Per dimostrare che un enunciato L relativo a un ciclo è corretto, definiamo L in termini di una serie di enunciati più brevi, L_0, L_1, \dots, L_k , dove:

1. L'enunciato *iniziale*, L_0 , è vero prima che il ciclo inizi.
2. Se L_{j-1} è vero prima dell'iterazione j , allora L_j sarà vero al termine dell'iterazione j .
3. Il fatto che l'enunciato finale, L_k , sia vero implica che l'enunciato da dimostrare, L , sia vero.

Vediamo un semplice esempio dell'uso di una condizione invariante in un ciclo per dimostrare la correttezza di un algoritmo. In particolare, usiamo questa tecnica per dimostrare che il metodo `arrayFind` (riportato nel Codice 4.11) trova effettivamente il minimo indice in corrispondenza del quale l'elemento `val` è presente all'interno dell'array `A`.

Codice 4.11: L'algoritmo `arrayFind` cerca e restituisce il minimo indice in corrispondenza del quale è presente, in un array, l'elemento cercato.

```

1  /** Restituisce il minimo indice j tale che data[j] == val, o -1 se val non c'è. */
2  public static int arrayFind(int[] data, int val) {
3      int n = data.length;
4      int j = 0;
5      while (j < n) { // val è diverso da tutti i primi j elementi di data
6          if (data[j] == val)
7              return j; // trovata una corrispondenza con l'indice j
8          j++; // procede con il prossimo indice
9          // val è diverso da tutti i primi j elementi di data
10     }
11     return -1; // se siamo arrivati qui, val non è presente nell'array data
12 }
```

Per dimostrare che `arrayFind` è corretto, definiamo induttivamente una sequenza di enunciati, L_j , che ci porterà alla correttezza dell'algoritmo. In particolare, affermiamo che all'inizio della j -esima iterazione del ciclo `while` è vero quanto segue:

L_j : `val` è diverso da tutti i primi j elementi di `data`.

Questa affermazione è vera all'inizio della prima iterazione del ciclo, perché j vale zero e, quindi, non c'è alcun elemento tra "i primi zero elementi" di `data` che possa essere uguale a `val` (spesso si dice che questo tipo di affermazione è *banalmente* vera). Durante l'esecuzione dell'iterazione j -esima, confrontiamo `val` con l'elemento `data[j]`: se sono equivalenti, resti-

tuiamo l'indice j , che è chiaramente corretto, perché nessun precedente elemento di `data` è uguale a `val`; se, invece, `val` e `data[j]` sono diversi, abbiamo trovato un ulteriore elemento diverso da `val` e possiamo incrementare l'indice j . Di conseguenza, l'affermazione L_j sarà vera anche per il nuovo valore assunto da j e, quindi, anche all'inizio dell'iterazione successiva. Se il ciclo termina senza aver restituito un indice, allora sarà $j = n$, per cui anche L_n è vera: non c'è nessun elemento di `data` uguale a `val`. Per concludere, l'algoritmo correttamente restituisce -1 , per segnalare che `val` non è presente in `data`.

4.5 Esercizi

Riepilogo e approfondimento

- R-4.1 Tracciare il grafico cartesiano delle funzioni $8n$, $4n \log n$, $2n^2$, n^3 e 2^n , usando una scala logaritmica per entrambi gli assi, x e y ; in pratica, se z è il valore di $f(n)$, rappresentare nel grafico un punto la cui coordinata x è $\log n$ e la cui coordinata y è $\log z$.
- R-4.2 Il numero di operazioni eseguite dagli algoritmi *A* e *B* è, rispettivamente, $8n \log n$ e $2n^2$. Determinare il valore n_0 tale che *A* sia migliore di *B* per $n \geq n_0$.
- R-4.3 Il numero di operazioni eseguite dagli algoritmi *A* e *B* è, rispettivamente, $40n^2$ e $2n^3$. Determinare il valore n_0 tale che *A* sia migliore di *B* per $n \geq n_0$.
- R-4.4 Fornire un esempio di funzione il cui grafico sia identico tanto nella scala log-log quanto nella scala normale.
- R-4.5 Spiegare perché, in scala log-log, il grafico della funzione n^c è una retta con pendenza c .
- R-4.6 Esprimere la somma di tutti i numeri pari che vanno da 0 a $2n$ in funzione del numero intero $n \geq 1$.
- R-4.7 Dimostrare che i due enunciati seguenti sono equivalenti:
- Il tempo d'esecuzione dell'algoritmo *A* è sempre $O(f(n))$.
 - Nel caso peggiore, il tempo d'esecuzione dell'algoritmo *A* è $O(f(n))$.
- R-4.8 Ordinare le funzioni seguenti in base al loro andamento asintotico.

$4n \log n + 2n$	2^{10}	$2^{\log n}$
$3n + 100 \log n$	$4n$	2^n
$n^2 + 10n$	n^3	$n \log n$

- R-4.9 Fornire una caratterizzazione mediante O-grande, in funzione di n , del tempo d'esecuzione del metodo `example1` del Codice 4.12 della pagina seguente.
- R-4.10 Fornire una caratterizzazione mediante O-grande, in funzione di n , del tempo d'esecuzione del metodo `example2` del Codice 4.12 della pagina seguente.
- R-4.11 Fornire una caratterizzazione mediante O-grande, in funzione di n , del tempo d'esecuzione del metodo `example3` del Codice 4.12 della pagina seguente.
- R-4.12 Fornire una caratterizzazione mediante O-grande, in funzione di n , del tempo d'esecuzione del metodo `example4` del Codice 4.12 della pagina seguente.
- R-4.13 Fornire una caratterizzazione mediante O-grande, in funzione di n , del tempo d'esecuzione del metodo `example5` del Codice 4.12 della pagina seguente.

Codice 4.12: Alcuni algoritmi da analizzare.

```

1  /** Restituisce la somma dei numeri interi presenti nell'array ricevuto. */
2  public static int example1(int[] arr) {
3      int n = arr.length, total = 0;
4      for (int j=0; j < n; j++)           // ciclo da 0 a n-1
5          total += arr[j];
6      return total;
7  }
8
9  /** Restituisce la somma dei numeri interi aventi indice pari nell'array ricevuto. */
10 public static int example2(int[] arr) {
11     int n = arr.length, total = 0;
12     for (int j=0; j < n; j += 2)        // notare l'incremento di 2
13         total += arr[j];
14     return total;
15 }
16
17 /** Restituisce la somma delle somme dei precedenti nell'array ricevuto. */
18 public static int example3(int[] arr) {
19     int n = arr.length, total = 0;
20     for (int j=0; j < n; j++)           // ciclo da 0 a n-1
21         for (int k=0; k <= j; k++)       // ciclo da 0 a j
22             total += arr[j];
23     return total;
24 }
25
26 /** Restituisce la somma delle somme dei precedenti nell'array ricevuto. */
27 public static int example4(int[] arr) {
28     int n = arr.length, prefix = 0, total = 0;
29     for (int j=0; j < n; j++) {           // ciclo da 0 a n-1
30         prefix += arr[j];
31         total += prefix;
32     }
33     return total;
34 }
35
36 /** Restituisce quanti valori di second sono = alle somme dei precedenti di first. */
37 public static int examples(int[] first, int[] second) {
38     int n = first.length, count = 0;
39     for (int i=0; i < n; i++) {           // ciclo da 0 a n-1
40         int total = 0;
41         for (int j=0; j < n; j++)         // ciclo da 0 a n-1
42             for (int k=0; k <= j; k++)       // ciclo da 0 a j
43                 total += first[k];
44             if (second[i] == total) count++;
45     }
46     return count;
47 }
```

R-4.14 Dimostrare che, se $d(n)$ è $O(f(n))$, allora $ad(n)$ è $O(f(n))$, per qualunque costante $a > 0$.

R-4.15 Dimostrare che, se $d(n)$ è $O(f(n))$ e $e(n)$ è $O(g(n))$, allora il prodotto $d(n)e(n)$ è $O(f(n)g(n))$.

R-4.16 Dimostrare che, se $d(n)$ è $O(f(n))$ e $e(n)$ è $O(g(n))$, allora $d(n) + e(n)$ è $O(f(n) + g(n))$.

R-4.17 Dimostrare che, se $d(n)$ è $O(f(n))$ e $e(n)$ è $O(g(n))$, allora non necessariamente $d(n) -$

$e(n)$ è $O(f(n) - g(n))$.

R-4.18 Dimostrare che, se $d(n)$ è $O(f(n))$ e $f(n)$ è $O(g(n))$, allora $d(n)$ è $O(g(n))$.

R-4.19 Dimostrare che $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$.

R-4.20 Dimostrare che $f(n)$ è $O(g(n))$ se e solo se $g(n)$ è $\Omega(f(n))$.

R-4.21 Dimostrare che, se $p(n)$ è una funzione polinomiale in n , allora $\log p(n)$ è $O(\log n)$.

R-4.22 Dimostrare che $(n+1)^5$ è $O(n^5)$.

R-4.23 Dimostrare che 2^{n+1} è $O(2^n)$.

R-4.24 Dimostrare che n è $O(n \log n)$.

R-4.25 Dimostrare che n^2 è $\Omega(n \log n)$.

R-4.26 Dimostrare che $n \log n$ è $\Omega(n)$.

R-4.27 Dimostrare che, se $f(n)$ è una funzione non decrescente che assume valori sempre maggiori di 1, allora $\lceil f(n) \rceil$ è $O(f(n))$.

R-4.28 Per ogni funzione $f(n)$ e per ogni tempo t presenti nella tabella seguente, determinare la massima dimensione n di un problema P che può essere risolto nel tempo t se l'algoritmo che risolve P richiede $f(n)$ microsecondi (un valore è già presente nella tabella).

	1 secondo	1 ora	1 mese	1 secolo
$\log n$	$\approx 10^{300000}$			
n				
$n \log n$				
n^2				
2^n				

R-4.29 L'algoritmo A esegue un calcolo che richiede un tempo d'esecuzione $O(\log n)$ per ogni elemento di un array che contiene n elementi. Qual è il tempo d'esecuzione di A nel caso peggiore?

R-4.30 Dato un array X con n elementi, l'algoritmo B sceglie $\log n$ elementi da X in modo casuale e, per ciascuno di essi, esegue un calcolo che richiede un tempo $O(n)$. Qual è il tempo d'esecuzione di B nel caso peggiore?

R-4.31 Dato un array X di numeri interi con n elementi, per ogni numero pari presente in X l'algoritmo C esegue un calcolo che richiede un tempo $O(n)$, mentre per ogni numero dispari presente in X esegue un calcolo che richiede un tempo $O(\log n)$. Qual è il tempo d'esecuzione di C nel caso peggiore e nel caso migliore?

R-4.32 Dato un array X con n elementi, l'algoritmo D invoca l'algoritmo E per ogni elemento $X[i]$, e l'algoritmo E richiede un tempo d'esecuzione $O(i)$ quando viene invocato con l'elemento $X[i]$. Qual è il tempo d'esecuzione di D nel caso peggiore?

R-4.33 Al e Bob stanno discutendo dei loro algoritmi. Al afferma che il suo metodo, che richiede un tempo d'esecuzione $O(n \log n)$, è *sempre* più veloce del metodo di Bob, che richiede un tempo $O(n^2)$. Per trovare il bandolo della matassa, eseguono un insieme di esperimenti. Con grande disappunto di Al, scoprono che, se $n < 100$, l'algoritmo $O(n^2)$ viene eseguito più velocemente e soltanto quando $n \geq 100$

l'algoritmo $O(n \log n)$ risulta migliore. Spiegare come questo sia possibile.

- R-4.34 Si dice che esista una città, che è ben nota anche se qui non verrà nominata, i cui abitanti gradiscono il proprio pranzo soltanto quando è il migliore che abbiano mai mangiato nella loro vita. Nell'ipotesi che la qualità dei pranzi sia distribuita in modo uniforme durante la vita di una persona, descrivere il numero atteso di volte in cui gli abitanti di questa città gradiranno il loro pranzo.

Creatività

- C-4.35 Nell'ipotesi che sia possibile ordinare n numeri in un tempo $O(n \log n)$, dimostrare che è possibile risolvere il problema dell'intersezione vuota tra tre insiemi in un tempo $O(n \log n)$.
- C-4.36 Descrivere un algoritmo efficiente per trovare i dieci elementi di valore maggiore in un array di dimensione n . Qual è il tempo d'esecuzione dell'algoritmo?
- C-4.37 Fornire un esempio di una funzione $f(n)$ positiva (cioè che assume valori sempre positivi) tale che $f(n)$ non sia né $O(n)$ né $\Omega(n)$.
- C-4.38 Dimostrare che $\sum_{i=1}^n i^2$ è $O(n^3)$.
- C-4.39 Dimostrare che $\sum_{i=1}^n i/2^i < 2$.
- C-4.40 Determinare il numero totale di chicchi di riso richiesti dall'inventore del gioco degli scacchi.
- C-4.41 Dimostrare che, se $b > 1$ è una costante, allora $\log_b f(n)$ è $\Theta(\log f(n))$.
- C-4.42 Descrivere un algoritmo che trovi tanto il minimo quanto il massimo di un insieme di n numeri usando un numero di confronti inferiore a $3n/2$.
- C-4.43 Bob ha creato un sito web di cui ha dato l'URL (cioè l'indirizzo) soltanto ai suoi n amici, che ha numerato da 1 a n . Ha detto all'amico i che può visitare il sito al massimo i volte. Bob dispone di un contatore, C , che tiene traccia del numero totale di visite al sito (senza, però, identificare il visitatore). Qual è il minimo valore di C che consente a Bob di sapere con certezza che uno dei suoi amici ha raggiunto il suo massimo numero di visite?
- C-4.44 Dimostrare in modo grafico la Proposizione 4.3, in analogia a quanto fatto nella Figura 4.3(b), nel caso in cui n sia dispari.
- C-4.45 L'array A contiene $n - 1$ numeri interi distinti appartenenti all'intervallo $[0, n - 1]$, per cui un solo numero di tale intervallo non è presente in A . Progettare un algoritmo che, in un tempo $O(n)$, scopra quel numero. Oltre all'array stesso, è consentito l'utilizzo di uno spazio addizionale che sia $O(1)$.
- C-4.46 Fare l'analisi asintotica dell'algoritmo di ordinamento per inserimento visto nel Paragrafo 3.1.2. Qual è il suo tempo d'esecuzione nel caso peggiore e nel caso migliore?
- C-4.47 Nelle reti di calcolatori, la sicurezza delle comunicazioni è estremamente importante e molti protocolli di rete la garantiscono usando la cifratura dei messaggi. Molti schemi crittografici per rendere sicure le trasmissioni su questi tipi di reti sono basati sul fatto che non esistano algoritmi efficienti per scomporre in fattori numeri interi molto grandi. Quindi, se siamo in grado di rappresentare un messaggio segreto mediante un numero primo p molto grande, possiamo trasmettere sulla rete il numero $r = p \cdot q$, dove $q > p$ è un altro numero primo che agisce da chiave di cifratura (encryption key). Un intercettatore che fosse in grado di ottenere il numero r trasmesso sulla rete dovrebbe essere in grado di scomporlo in fattori per poter

ricavare il messaggio segreto p .

Ricavare un messaggio mediante la scomposizione in fattori, senza conoscere la chiave di cifratura q , è difficile. Per capire perché, consideriamo il seguente banale algoritmo di scomposizione in fattori:

```
for (int p=2; p < r; p++)
    if (r % p == 0)
        return "The secret message is p!";
```

- Supponiamo che il computer dell'intercettatore sia in grado di dividere due numeri interi rappresentati con 100 bit in un microsecondo (cioè effettua un milione di divisioni al secondo). Stimare il tempo richiesto, nel caso peggiore, per decifrare il messaggio segreto p se il messaggio trasmesso r è rappresentato con 100 bit.
- Qual è la complessità temporale dell'algoritmo qui presentato, nel caso peggiore? Poiché il dato fornito in ingresso all'algoritmo è soltanto il numero r , si ipotizzi che la dimensione del problema, n , sia il numero di byte necessari per memorizzare r , cioè $n = \lceil (\log_2 r) / 8 \rceil + 1$, e che ogni divisione richieda un tempo $O(n)$.

- C-4.48 Al afferma di poter dimostrare che tutte le pecore di un gregge sono dello stesso colore:

Caso base: Una pecora. Ha chiaramente lo stesso colore di se stessa.

Passo induttivo: Un gregge di n pecore. Togliamo dal gregge una pecora, a . Le restanti $n - 1$ pecore sono tutte dello stesso colore, per ipotesi induttiva. Ora, rimettiamo la pecora a nel gregge e togliamo una diversa pecora, b . Sempre per ipotesi induttiva, le $n - 1$ pecore rimaste nel gregge (tra le quali c'è a) sono tutte dello stesso colore, che non può essere altro che il colore precedente, dato che $n - 2$ pecore sono le stesse di prima. Quindi, anche a è di quello stesso colore, per cui tutte le pecore del gregge di dimensione n sono dello stesso colore.

Cosa c'è di sbagliato nella "dimostrazione" di Al?

- C-4.49 Analizzare la seguente "dimostrazione" del fatto che la funzione di Fibonacci (definita nella Proposizione 4.20) sia $O(n)$:

Caso base: $n \leq 2$. $F(1) = 1$ e $F(2) = 2$.

Passo induttivo: $n > 2$. Ipotizziamo che l'affermazione sia vera per $n' < n$ e consideriamo n . $F(n) = F(n-2) + F(n-1)$. Per ipotesi induttiva, $F(n-2)$ è $O(n-2)$ e $F(n-1)$ è $O(n-1)$. Quindi, $F(n)$ è $O((n-2) + (n-1))$, per l'identità dimostrata nell'Esercizio R-4.16. Da questo discende che $F(n)$ è $O(n)$.

Cosa c'è di sbagliato in questa "dimostrazione"?

- C-4.50 Dimostrare per induzione che la funzione di Fibonacci (definita nella Proposizione 4.20) è $\Omega((3/2)^n)$.

- C-4.51 Sia S un insieme di n rette in un piano tali che non esista una coppia di rette parallele e non esistano tre rette passanti per uno stesso punto. Dimostrare, per induzione, che le rette di S danno luogo complessivamente a $\Theta(n^2)$ punti di intersezione.

C-4.52 Dimostrare che la sommatoria $\sum_{i=1}^n \log i$ è $O(n \log n)$.

C-4.53 Dimostrare che la sommatoria $\sum_{i=1}^n \log i$ è $\Omega(n \log n)$.

C-4.54 Sia $p(x)$ una funzione polinomiale di grado n , cioè $p(x) = \sum_{i=0}^n a_i x^i$.

- Descrivere un semplice algoritmo che calcoli $p(x)$ in un tempo $O(n^2)$.
- Descrivere un algoritmo che calcoli $p(x)$ in un tempo $O(n \log n)$, basandosi su un calcolo più efficiente di x^i .
- Si immagini, ora, di riscrivere $p(x)$ in questo modo:

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + x a_n) \cdots))),$$

utilizzando il cosiddetto *metodo di Horner*. Usando la notazione O-grande, caratterizzare il numero di operazioni aritmetiche eseguite da questo metodo di calcolo di $p(x)$.

C-4.55 Un malefico re possiede n bottiglie di vino e un cospiratore ne ha avvelenata una: sfortunatamente nessuno sa quale sia. Il veleno è molto pericoloso: una sola goccia, anche se diluita in proporzione un miliardo a uno, è letale. Ciò nonostante, deve trascorrere un intero mese prima che il veleno faccia effetto. Progettare un algoritmo che consenta di determinare in modo certo, in un solo mese di tempo, quale sia l'unica bottiglia avvelenata, sfruttando la "collaborazione" di un numero $O(\log n)$ di assaggiatori.

C-4.56 L'array A contiene n numeri interi, anche non tutti distinti, appartenenti all'intervallo $[0, 4n]$. Descrivere un algoritmo efficiente per determinare il numero intero k che ricorre più frequentemente in A . Qual è il tempo d'esecuzione dell'algoritmo?

C-4.57 Dato l'array A contenente n numeri interi positivi, ciascuno dei quali è rappresentato con $k = \lceil \log n \rceil + 1$ bit, descrivere un metodo che in un tempo $O(n)$ individui un numero intero di k bit che non sia presente in A .

C-4.58 Dimostrare che qualunque soluzione del problema precedente necessita di un tempo d'esecuzione $\Omega(n)$.

C-4.59 Dato l'array A contenente n numeri interi qualsiasi, progettare un metodo che, in un tempo $O(n)$, individui un numero intero che non possa essere ottenuto come somma di due numeri presenti in A .

Progettazione

P-4.30 Effettuare un'analisi sperimentale dei due algoritmi, `prefixAverage1` e `prefixAverage2`, visti nel Paragrafo 4.3.3. Visualizzare i loro tempi d'esecuzione in funzione della dimensione dei dati in ingresso usando un grafico log-log.

P-4.31 Effettuare un'analisi sperimentale che confronti i tempi d'esecuzione relativi dei metodi riportati nel Codice 4.12.

P-4.32 Effettuare un'analisi sperimentale che verifichi l'ipotesi secondo cui il metodo `Arrays.sort` della libreria di Java richiede, in media, un tempo d'esecuzione $O(n \log n)$.

P-4.33 Per ciascuno dei due algoritmi, `unique1` e `unique2`, che risolvono il problema dell'unicità degli elementi di un array, effettuare un'analisi sperimentale che determini il massimo valore di n che consente l'esecuzione dell'algoritmo in un

tempo non superiore a un minuto.

Note

Molti sono stati i commenti relativi a un utilizzo appropriato della notazione O-grande [18, 43, 59]. Knuth [60, 59] l'ha definita usando la notazione $f(n) = O(g(n))$, dicendo però che tale "uguaglianza" andava intesa come valida "in una sola direzione". In questo libro abbiamo scelto di seguire un concetto di uguaglianza più normale e abbiamo considerato la notazione O-grande come un insieme, coerentemente con Brassard [18]. Il lettore interessato ad approfondire l'analisi del caso medio faccia riferimento al lavoro di Vitter e Flajolet [93].

5

Ricorsione

Per descrivere le ripetizione di azioni in un programma per calcolatore si possono usare i cicli, come i costrutti Java `while` e `for` descritti nel Paragrafo 1.5.2. Un modo completamente diverso di ottenere l'esecuzione ripetuta di sezioni di codice sfrutta un procedimento che prende il nome di **ricorsione** (*recursion*).

La ricorsione è una tecnica che consente a un metodo di effettuare una o più invocazioni di se stesso durante la propria esecuzione, oppure consente a una struttura dati di basare la propria rappresentazione su esemplari più piccoli dello stesso tipo di struttura. Tanto l'arte quanto la natura presentano molti esempi di ricorsione: ad esempio, gli schemi frattali sono ricorsivi per natura. Un esempio concreto di ricorsione usato nell'arte si riscontra nelle bambole russe chiamate *matrioska*: ciascuna bambola è fatta di legno e, tranne la più piccola che è di legno massiccio, è cava e contiene un'altra bambola al proprio interno.

In informatica, la ricorsione costituisce un'alternativa elegante e potente per eseguire compiti ripetitivi. Infatti, alcuni linguaggi di programmazione (pochi, effettivamente, come Scheme e Smalltalk) non forniscono un supporto esplicito alla definizione di strutture iterative, come i cicli, ma si basano direttamente sulla ricorsione per poter esprimere il concetto di ripetizione. La maggior parte dei linguaggi di programmazione moderni consente la ricorsione funzionale, mediante un meccanismo identico a quello utilizzato per eseguire le normali invocazioni di metodi. Quando l'invocazione di un metodo è, in realtà, un'invocazione ricorsiva, questa viene sospesa finché l'intera ricorsione non giunge al termine.

La ricorsione è una tecnica veramente importante per lo studio delle strutture dati e degli algoritmi e la useremo soprattutto in alcuni capitoli del libro (in particolare, nei Capitoli 8 e 12). In questo capitolo iniziamo da quattro esempi che illustrano l'uso della ricorsione, fornendo anche la loro implementazione Java.

- La **funzione fattoriale** (solitamente indicata con $n!$) è una ben nota funzione matematica che ha una definizione naturalmente ricorsiva.
- Un **righello in pollici** ha uno schema ricorsivo che è un semplice esempio di struttura frattale.
- La **ricerca binaria** è uno degli algoritmi più importanti dell'informatica: consente di trovare in modo efficiente un valore all'interno di un insieme di dati, anche di dimensioni enormi.
- Il **file system** ("sistema di gestione dei file archiviati") di un calcolatore ha una struttura ricorsiva, perché le sue cartelle (o *directory*) possono essere arbitrariamente annidate all'interno di altre cartelle, anche su molti livelli di profondità. Per gestire tali *file system* e per navigare al loro interno vengono diffusamente utilizzati algoritmi ricorsivi.

Passeremo poi a descrivere come si esegua un'analisi formale del tempo d'esecuzione di un algoritmo ricorsivo, discutendo anche di alcune potenziali trappole in cui è facile cadere quando si definiscono algoritmi o strutture usando la ricorsione.

5.1 Esempi di ricorsione

5.1.1 Funzione fattoriale

Per illustrare il meccanismo di funzionamento della ricorsione, cominciamo da un semplice esempio matematico, il calcolo del valore della **funzione fattoriale**. Il fattoriale di un numero intero positivo n , che si indica con $n!$, è definito come il prodotto dei numeri interi che vanno da 1 a n . Se $n = 0$, si definisce $n!$ uguale a 1, per convenzione. In modo più formale, per qualsiasi numero intero $n \geq 0$:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{se } n \geq 1 \end{cases}$$

Ad esempio, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. La funzione fattoriale è importante, ad esempio, perché si sa che $n!$ è il numero di diverse disposizioni in sequenza di n elementi diversi, cioè il numero di **permutazioni** di n elementi distinti. Ad esempio, i tre caratteri a, b e c possono essere disposti in sequenza in $3! = 3 \cdot 2 \cdot 1 = 6$ modi diversi: abc, acb, bac, bca, cab e cba.

Per la funzione fattoriale esiste anche una definizione naturalmente ricorsiva. Per ottenerla, osserviamo che $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$. Più in generale, per qualunque numero intero positivo n , si può definire $n!$ come $n \cdot (n-1)!$. Questa **definizione ricorsiva** può essere formalizzata così:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n \geq 1 \end{cases}$$

Questo schema di definizione è simile a molte altre definizioni ricorsive di funzioni. Innanzitutto, abbiamo **uno o più casi base**, che si riferiscono a valori prefissati della funzione. La definizione precedente ha un solo caso base, che afferma che $n! = 1$ per $n = 0$. Poi, troviamo

uno o più *casi ricorsivi*, che definiscono la funzione in termini di se stessa. Nella definizione della funzione ricorsiva c'è un unico caso, secondo il quale $n! = n \cdot (n - 1)!$ per $n \geq 1$.

Un'implementazione ricorsiva della funzione fattoriale

La ricorsione non è solamente una notazione matematica: possiamo usarla per progettare un'implementazione, in Java, della funzione fattoriale, come si può vedere nel Codice 5.1.

Codice 5.1: Un'implementazione ricorsiva della funzione fattoriale.

```

1 public static int factorial(int n) throws IllegalArgumentException {
2     if (n < 0)
3         throw new IllegalArgumentException(); // n non può essere negativo
4     else if (n == 0)
5         return 1;                                // caso base
6     else
7         return n * factorial(n - 1);           // caso ricorsivo
8 }
```

Questo metodo non usa alcun ciclo esplicito: si ottiene l'iterazione mediante ripetute invocazioni ricorsive del metodo. Il processo è finito perché, ogni volta che il metodo viene invocato, il suo parametro è inferiore di un'unità rispetto all'invocazione precedente e, quando si raggiunge il caso base, non vengono più effettuate invocazioni ricorsive.

Per illustrare l'esecuzione di un metodo ricorsivo usiamo un *diagramma di ricorsione* (*recursion trace*). Ogni riquadro presente nel diagramma corrisponde a un'invocazione ricorsiva e ogni nuova invocazione ricorsiva del metodo viene rappresentata da una freccia rivolta verso il basso, verso una nuova invocazione. Quando il metodo termina la propria esecuzione e restituisce un valore, l'evento viene rappresentato nel diagramma mediante un'arretrofrecchia che torna verso l'alto e il valore restituito è segnalato accanto alla freccia. La Figura 5.1 riporta un esempio di tale diagramma per l'esecuzione della funzione fattoriale.

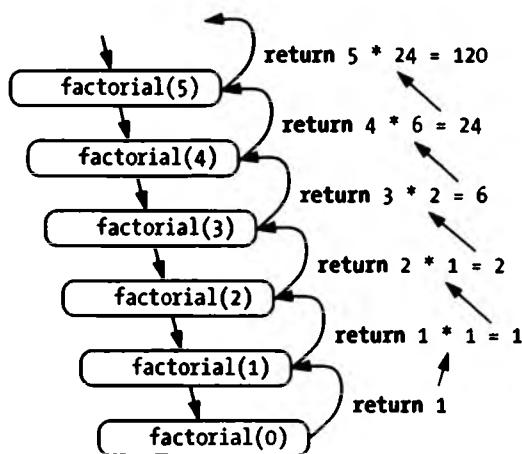


Figura 5.1: Un diagramma che rappresenta l'esecuzione ricorsiva dell'invocazione factorial(5).

Un diagramma di ricorsione riflette molto fedelmente l'esecuzione della ricorsione così come espressa in un linguaggio di programmazione. In Java, ogni volta che viene invocato un

metodo (ricorsivo oppure no), viene creata una struttura che ne contiene i *dati di attivazione* (*activation record* o *activation frame*) e memorizza informazioni relative a quella particolare invocazione di quel metodo. Tali dati contengono i parametri e le variabili locali specifici di una data invocazione del metodo, oltre a informazioni relative all'enunciato attualmente in esecuzione all'interno del corpo del metodo.

Quando l'esecuzione del metodo porta a un'invocazione annidata di un altro metodo (o dello stesso metodo), l'esecuzione del metodo attualmente attivo viene sospesa e nei suoi dati di attivazione viene aggiunta un'indicazione del punto del codice in cui il controllo del flusso d'esecuzione dovrà riprendere dopo che l'invocazione annidata sarà terminata. Quindi, vengono creati i nuovi dati di attivazione, per la nuova invocazione annidata. Questa procedura avviene tanto per un'invocazione normale, dove un metodo invoca un metodo diverso, quanto per un'invocazione ricorsiva, dove un metodo invoca se stesso. La cosa importante è che ci sia un diverso insieme di dati di attivazione per ogni invocazione attiva.

5.1.2 Disegnare un righello in pollici

Nel caso del calcolo della funzione fattoriale, non c'è nessun motivo che spinga a preferire la ricorsione rispetto all'iterazione diretta con un ciclo. Come esempio più complesso di utilizzo della ricorsione, vediamo ora come si possano disegnare i segmenti numerati che costituiscono un classico righello in pollici. Per ogni pollice, dobbiamo posizionare un piccolo segmento con un'etichetta numerica. Chiamiamo *lunghezza maggiore* (*major tick length*) la lunghezza dei segmenti che indicano un numero intero di pollici, mentre i segmenti più corti che si trovano tra due segmenti lunghi avranno una *lunghezza minore* (*minor tick length*) e saranno posizionati ogni mezzo pollice, ogni quarto di pollice, e così via. Ogni volta che la dimensione dell'intervallo misurato si dimezza, la lunghezza del segmento corrispondente diminuisce di un'unità. La Figura 5.2 mostra alcuni di questi righelli con varie misure della lunghezza maggiore (anche se non sono disegnati in scala).

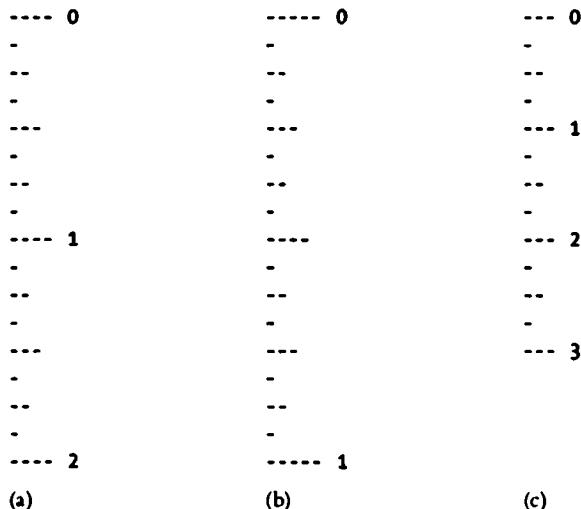


Figura 5.2: Tre esempi di righelli in pollici: (a) un righello di 2 pollici con lunghezza maggiore uguale a 4; (b) un righello di 1 pollice con lunghezza maggiore uguale a 5; (c) un righello di 3 pollici con lunghezza maggiore uguale a 3.

Un approccio ricorsivo al disegno di un righello

Lo schema di un righello in pollici è un semplice esempio di *frattale*, cioè di una forma geometrica che ha una struttura auto-ricorsiva a vari livelli di ingrandimento. Analizziamo il righello avente lunghezza maggiore uguale a 5, nella Figura 5.2(b). Ignorando i segmenti che contengono 0 e 1, vediamo come si possa disegnare la sequenza di segmenti che si trova tra i due citati. Il segmento centrale (in corrispondenza di mezzo pollice) ha lunghezza 4 e osserviamo che i due schemi di segmenti sopra e sotto questo segmento centrale sono identici, ciascuno con un segmento centrale di lunghezza 3.

In generale, una sezione di righello con segmento centrale di lunghezza $L \geq 1$ è composta da:

- Una sezione con un segmento centrale di lunghezza $L - 1$
- Un segmento di lunghezza L
- Una sezione con un segmento centrale di lunghezza $L - 1$

Sebbene sia possibile disegnare tale righello usando un procedimento iterativo (si veda l'Esercizio P-5.29), è molto più semplice farlo usando la ricorsione. La nostra implementazione è costituita da quattro metodi, come si può vedere nel Codice 5.2.

Codice 5.2: Un'implementazione ricorsiva del metodo che disegna un righello.

```

1  /** Disegna un righello di data lunghezza e con data lunghezza maggiore. */
2  public static void drawRuler(int nInches, int majorLength) {
3      drawLine(majorLength, 0);           // disegna il segmento 0 con la sua etichetta
4      for (int j=1; j <= nInches; j++) {
5          drawInterval(majorLength - 1); // disegna i segmenti interni al pollice j
6          drawLine(majorLength, j);      // disegna il segmento j con la sua etichetta
7      }
8  }
9  private static void drawInterval(int centralLength) {
10     if (centralLength >= 1) {           // altrimenti non fa nulla
11         drawInterval(centralLength - 1); // disegna ricorsivamente la sezione superiore
12         drawLine(centralLength);       // disegna il segmento centrale (senza etichetta)
13         drawInterval(centralLength - 1); // disegna ricorsivamente la sezione inferiore
14     }
15 }
16 private static void drawLine(int tickLength, int tickLabel) {
17     for (int j=0; j < tickLength; j++)
18         System.out.print("-");
19     if (tickLabel >= 0)
20         System.out.print(" " + tickLabel);
21     System.out.print("\n");
22 }
23 /** Disegna un segmento con la lunghezza data (senza etichetta). */
24 private static void drawLine(int tickLength) {
25     drawLine(tickLength, -1);
26 }
```

Il metodo principale, `drawRuler`, gestisce la costruzione dell'intero righello. I suoi argomenti specificano la lunghezza totale del righello, in pollici, e la *lunghezza maggiore* dei suoi segmenti. Il metodo ausiliario, `drawLine`, disegna un singolo segmento, di cui viene specificato

come parametro il numero di trattini che lo compongono (e un'etichetta facoltativa, che è un numero intero che viene scritto alla destra del segmento).

La parte di lavoro interessante viene svolta dal metodo ricorsivo `drawInterval`, che disegna la sequenza di segmenti presenti all'interno di una determinata sezione del righello, basandosi sulla lunghezza del segmento centrale della sezione. Il progetto è fatto seguendo lo schema generale individuato nella fase di analisi, con il caso base che riguarda il valore $L = 0$, che non disegna alcunché. Per $L \geq 1$, il primo e l'ultimo passo vengono eseguiti invocando ricorsivamente `drawInterval($L - 1$)`, mentre il passo intermedio richiede l'invocazione del metodo `drawLine(L)`.

Illustrare il disegno di un righello usando un diagramma di ricorsione

Possiamo visualizzare l'esecuzione del metodo ricorsivo `drawInterval` usando un diagramma di ricorsione, che è più complicato rispetto all'esempio visto per il fattoriale, perché ogni invocazione di `drawInterval` dà luogo a due ulteriori invocazioni ricorsive. Per illustrare questo fenomeno, mostreremo il diagramma di ricorsione in un formato che ricorda lo schema di un documento, come si può vedere nella Figura 5.3.

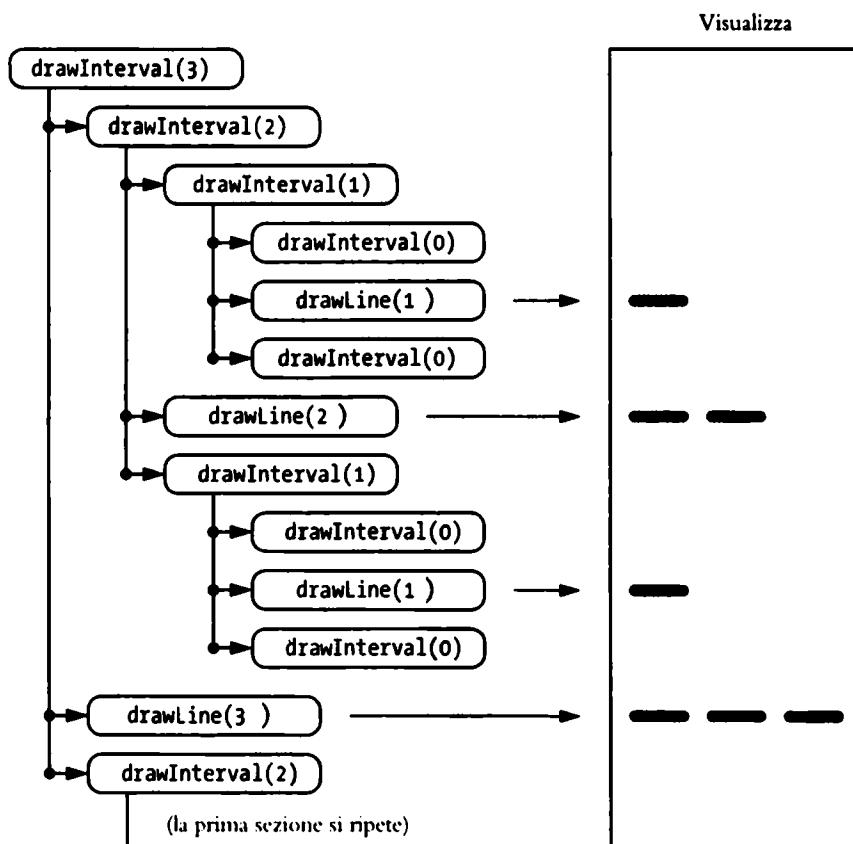


Figura 5.3: Una parte del diagramma di ricorsione per l'invocazione `drawInterval(3)`. La sezione contenente le invocazioni conseguenti alla seconda invocazione di `drawInterval(2)` non è mostrata, ma è identica alla prima sezione.

5.1.3 Ricerca binaria

In questo paragrafo descriveremo un classico algoritmo ricorsivo, la *ricerca binaria* o *ricerca per bisezione* (*binary search*), utilizzata per individuare in modo efficiente un valore desiderato all'interno di una sequenza ordinata di n elementi memorizzata in un array. Si tratta di uno dei più importanti algoritmi dell'informatica e la sua applicazione è il motivo principale per cui i dati vengono spesso conservati in ordine (come nella Figura 5.4).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Figura 5.4: Valori conservati in ordine all'interno di un array. I numeri in alto sono gli indici.

Quando la sequenza *non è ordinata* (*unsorted*), per cercare un valore al suo interno normalmente si usa un ciclo che esamina un elemento dopo l'altro, fin quando non viene trovato l'elemento cercato oppure si esaurisce l'insieme dei dati. Questo algoritmo prende il nome di *ricerca lineare* (*linear search*) o *ricerca sequenziale* (*sequential search*) e viene eseguito in un tempo $\mathcal{O}(n)$ (cioè un tempo *lineare*), perché, nel caso peggiore, ispeziona ciascun singolo elemento.

Quando la sequenza è *ordinata* (*sorted*) e i suoi elementi sono *accessibili in un tempo costante tramite un indice*, cioè c'è l'accesso casuale alla sequenza, esiste un algoritmo più efficiente (per capirlo, pensate a come risolvereste un simile problema a mano!). Se consideriamo un elemento arbitrario della sequenza avente valore v , siamo sicuri che tutti gli elementi che lo precedono nella sequenza ordinata hanno valori non maggiori di v e che tutti gli elementi che lo seguono hanno valori non minori di v . Questa osservazione ci consente di giungere rapidamente alla posizione dell'elemento cercato usando una variante di un gioco assai diffuso tra i bambini, "di più o di meno" (*high-low*). A un certo punto della ricerca, diciamo che un elemento della sequenza è un *candidato* se non siamo in grado di affermare che non possa essere l'elemento che stiamo cercando. L'algoritmo gestisce due parametri, *low* e *high*, tali che tutti gli elementi candidati presenti nell'array abbiano indici compresi tra *low* e *high*, compresi. Inizialmente *low* = 0 e *high* = $n - 1$, cioè tutti gli elementi dell'array sono candidati, perché non abbiamo ancora acquisito informazioni. Poi, confrontiamo il valore cercato (*target*, cioè "obiettivo" della ricerca) con il *candidato mediano*, cioè quell'elemento il cui indice vale:

$$\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor.$$

Consideriamo tre casi:

- Se il valore cercato è uguale al candidato mediano, la ricerca ha avuto successo e può terminare con esito positivo.
- Se il valore cercato è minore del candidato mediano, possiamo invocare ricorsivamente la ricerca sulla prima metà della sequenza, cioè sull'intervallo di indici che va da *low* a *mid* - 1.
- Se il valore cercato è maggiore del candidato mediano, possiamo invocare ricorsivamente la ricerca sulla seconda metà della sequenza, cioè sull'intervallo di indici che va da *mid* + 1 a *high*.

Se si arriva alla situazione in cui $\text{low} > \text{high}$, la ricerca non ha avuto successo, perché l'intervallo dei candidati, $[\text{low}, \text{high}]$, è vuoto.

Questo algoritmo è noto come *ricerca binaria o per bisezione*: il Codice 5.3 ne riporta l'implementazione in Java e la Figura 5.5 ne illustra l'esecuzione. Mentre la ricerca sequenziale viene eseguita in un tempo $O(n)$, la ricerca binaria, decisamente più efficiente, richiede un tempo $O(\log n)$. Si tratta di un miglioramento molto significativo, dal momento che, quando n vale un miliardo, $\log_2 n$ vale soltanto 30 (rimandiamo l'analisi formale del tempo d'esecuzione dell'algoritmo di ricerca binaria alla Proposizione 5.2, nel Paragrafo 5.2).

Codice 5.3: Un'implementazione dell'algoritmo di ricerca binaria in un array ordinato.

```

1  /**
2   * Restituisce true se e solo target si trova nella porzione di array indicata.
3   * La ricerca considera soltanto la porzione di array tra data[low] e data[high].
4   */
5  public static boolean binarySearch(int[] data, int target, int low, int high) {
6      if (low > high)
7          return false;           // intervallo vuoto, valore non trovato
8      else {
9          int mid = (low + high) / 2;
10         if (target == data[mid])
11             return true;        // valore trovato
12         else if (target < data[mid])
13             return binarySearch(data, target, low, mid - 1); // ricorsione metà sinistra
14         else
15             return binarySearch(data, target, mid + 1, high); // ricorsione metà destra
16     }
17 }
```

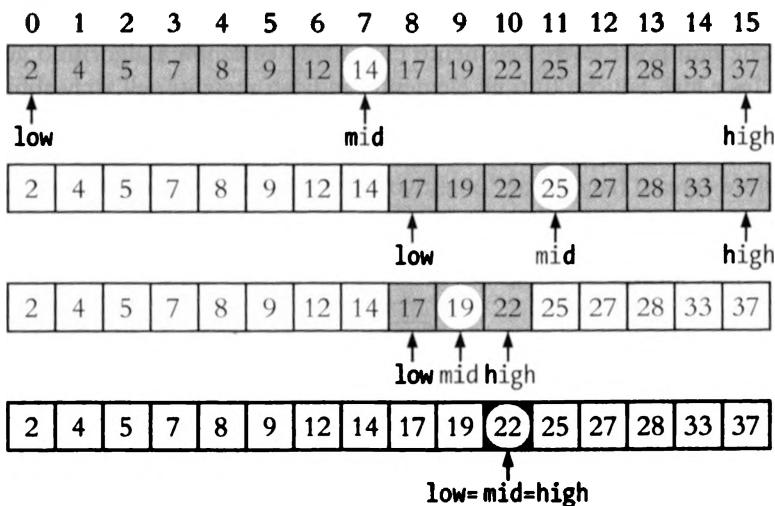


Figura 5.5: Esempio di una ricerca binaria relativa al valore 22 all'interno di un array ordinato con 16 elementi.

5.1.4 File system

I moderni sistemi operativi definiscono in modo ricorsivo le cartelle (dette anche *directory* o *folder*) all'interno dei *file system* (cioè dei sistemi di gestione degli archivi, o *file*). Un file system è costituito da una *cartella radice* (*top-level directory*), il cui contenuto è costituito da file e da altre cartelle, che a loro volta possono contenere file e altre cartelle, e così via. Il sistema operativo permette che le cartelle siano annidate dentro altre cartelle fino a una "profondità" arbitraria (almeno finché c'è spazio in memoria), anche se, ovviamente, è necessario che alla fine ci siano cartelle "di base" che contengono solamente file, senza ulteriori sottocartelle. La Figura 5.6 mostra una porzione di un tale file system.

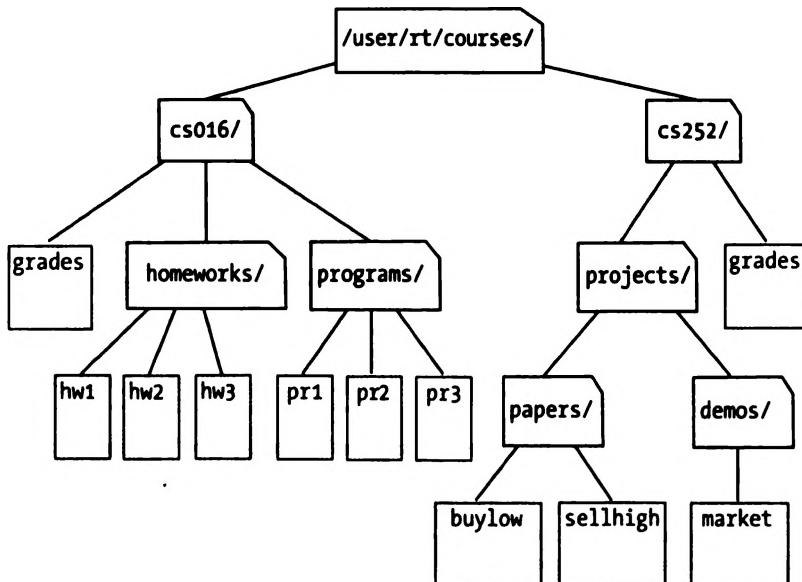


Figura 5.6: Una porzione di un file system che mostra con evidenza l'organizzazione gerarchica, con cartelle annidate dentro altre cartelle.

Vista la natura ricorsiva della rappresentazione del file system, non dovrebbe essere una sorpresa il fatto che molti comportamenti del sistema operativo, utilizzati assai frequentemente, come la copiatura o l'eliminazione di una cartella, siano realizzati mediante algoritmi ricorsivi. In questo paragrafo vediamo uno di tali algoritmi: il calcolo dello spazio totale utilizzato, all'interno del disco, per tutti i file e le cartelle annidate all'interno di una cartella specificata.

A titolo d'esempio, la Figura 5.7 mostra lo spazio utilizzato da tutte le entità definite nel file system che abbiamo usato come esempio nella Figura 5.6. Distinguiamo tra lo spazio *effettivo* utilizzato da ciascuna entità e lo spazio *cumulativo* occupato da quella entità e da tutti i file e cartelle annidati in essa. Ad esempio, la cartella cs016 usa solamente 2K di spazio effettivo, ma occupa cumulativamente uno spazio di 249K.

Lo spazio cumulativo occupato da un'entità può essere calcolato con un semplice algoritmo ricorsivo: è uguale allo spazio effettivamente utilizzato dall'entità sommato allo spazio cumulativo usato da ogni altra entità memorizzata direttamente nell'entità stessa,

se questa è una cartella. Ad esempio, lo spazio cumulativo imputabile alla cartella `cs016` è 249K perché 2K è lo spazio occupato direttamente dalla cartella, 8K è lo spazio cumulativo occupato da `grades`, 10K è lo spazio cumulativo occupato da `homeworks` e, infine, 229K è lo spazio cumulativo occupato da `programs`. Il Codice 5.4 mostra lo pseudocodice che descrive l'algoritmo.

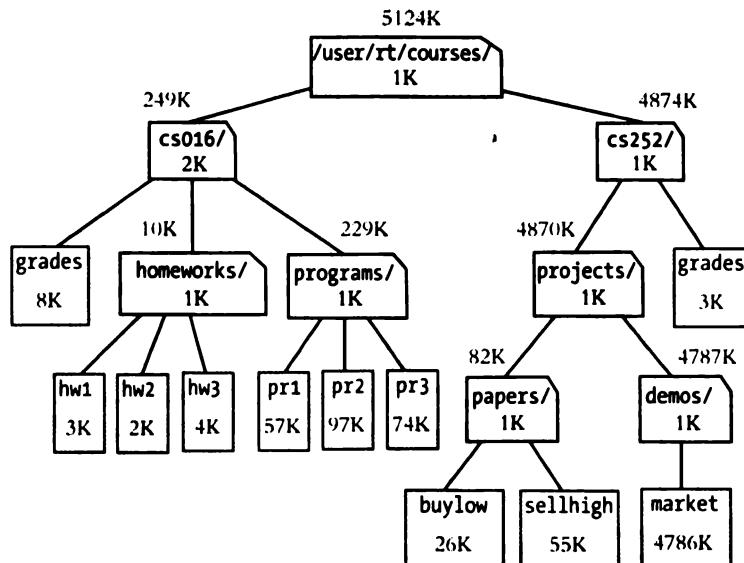


Figura 5.7: La stessa porzione di file system vista nella Figura 5.6, con l'aggiunta di annotazioni che descrivono la quantità di spazio utilizzato all'interno del disco. All'interno del rettangolo che rappresenta ciascun file e ciascuna cartella è indicato lo spazio effettivamente usato da tale entità, mentre al di sopra del rettangolo che raffigura ciascuna cartella è riportato lo spazio cumulativo occupato dalla cartella e da ciò che essa contiene (ricorsivamente).

Codice 5.4: Un algoritmo che calcola lo spazio cumulativo occupato sul disco da un'entità (file o cartella) di un file system. Ipotizziamo che il metodo `size` restituisca lo spazio effettivamente occupato da una di tali entità.

Algoritmo `DiskUsage(path)`:

Input: Una stringa che identifica il percorso, `path`, che conduce a un'entità in un file system

Output: Lo spazio cumulativo occupato sul disco da quell'entità e da tutte le entità ricorsivamente contenute al suo interno

```

total = size(percorso)           { spazio effettivamente utilizzato dall'entità }
if path rappresenta una cartella then
    for ogni entità child memorizzata nella cartella path do
        total = total + DiskUsage(child)      { invocazione ricorsiva }
    return total
  
```

La classe `java.io.File`

Per implementare in Java l'algoritmo ricorsivo che calcola lo spazio utilizzato su disco utiliziamo i servizi messi a disposizione dalla classe `java.io.File`. Un esemplare di questa classe rappresenta, in astratto, il nome di un percorso (*pathname*) all'interno del sistema operativo e consente di chiedere al sistema operativo informazioni relative alla risorsa associata a quel percorso. In particolare, di quella classe useremo i seguenti metodi:

- `new File(pathString)` oppure `new File(parentFile, childString)`

Un nuovo esemplare di `File` può essere costruito fornendo come parametro il suo percorso completo (`pathString`, sotto forma di stringa) oppure un esemplare esistente di `File` (`parentFile`) che rappresenta la cartella a cui appartiene il file e una stringa, `childString`, che ne rappresenta il nome all'interno di tale cartella.

- `file.length()`

Restituisci lo spazio effettivamente occupato sul disco (misurato in byte) dall'entità del file system rappresentata dall'esemplare file di tipo File (ad esempio, /user/rt/courses).

- `file.isDirectory()`

Restituisce `true` se e solo se l'entità del file system rappresentata dall'esemplare file di tipo `File` è una cartella.

- ### • file.list()

Restituisce un array di stringhe contenente i nomi di tutte le entità presenti all'interno della cartella rappresentata dall'esemplare file. Nel nostro esempio di file system, se invochiamo questo metodo con l'esemplare di tipo File associato al percorso /user/rt/courses/cs016, viene restituito un array con questo contenuto: {"grades", "homeworks", "programs"}.

Implementazione in Java

Usando la classe `File`, possiamo ora convertire l'algoritmo descritto nel Codice 5.4 nella sua implementazione in Java, riportata nel Codice 5.5.

Codice 5.5: Un metodo ricorsivo per il calcolo della spazio utilizzato all'interno di un file system.

Diagramma di ricorsione

Per fare in modo che sia l'esecuzione stessa del codice a produrre un diagramma di ricorsione, abbiamo inserito nella nostra implementazione in Java un enunciato di visualizzazione che non è strettamente necessario a produrre il risultato (riga 13 del Codice 5.5). Il formato dell'informazione visualizzata riproduce intenzionalmente quello prodotto dal programma standard di Unix/Linux du (sigla che sta per "disk usage", cioè occupazione del disco): la quantità di spazio utilizzato da una cartella e da tutte le entità contenute al suo interno. Il risultato può essere anche piuttosto lungo, come si può vedere nella Figura 5.8.

```

8      /user/rt/courses/cs016/grades
3      /user/rt/courses/cs016/homeworks/hw1
2      /user/rt/courses/cs016/homeworks/hw2
4      /user/rt/courses/cs016/homeworks/hw3
10     /user/rt/courses/cs016/homeworks
57     /user/rt/courses/cs016/programs/pr1
97     /user/rt/courses/cs016/programs/pr2
74     /user/rt/courses/cs016/programs/pr3
229    /user/rt/courses/cs016/programs
249    /user/rt/courses/cs016
26     /user/rt/courses/cs252/projects/papers/buylow
55     /user/rt/courses/cs252/projects/papers/sellhigh
82     /user/rt/courses/cs252/projects/papers
4786   /user/rt/courses/cs252/projects/demos/market
4787   /user/rt/courses/cs252/projects/demos
4870   /user/rt/courses/cs252/projects
3      /user/rt/courses/cs252/grades
4874   /user/rt/courses/cs252
5124   /user/rt/courses/

```

Figura 5.8: Spazio occupato nel disco dalla porzione di file system vista nella Figura 5.7, così come viene riportato dall'esecuzione del nostro metodo `diskUsage`, visto nel Codice 5.5, oppure, in modo equivalente, dal comando `du` di Unix/Linux usato con l'opzione `-a` (che elenca tanto le cartelle quanto i file).

Quando viene eseguita sull'esempio di file system riportato nella Figura 5.7, la nostra implementazione del metodo `diskUsage` produce il risultato visibile nella Figura 5.8. Durante l'esecuzione dell'algoritmo, viene effettuata una e soltanto una invocazione ricorsiva del metodo per ogni entità presente nella porzione di file system che viene presa in esame. Dato che ogni riga informativa viene visualizzata subito prima che termini una delle invocazioni, le righe visualizzate riflettono l'ordine in cui le invocazioni ricorsive *vengono portate a termine*. Si noti che lo spazio cumulativo relativo a un'entità annidata viene calcolato e visualizzato prima di calcolare e visualizzare lo spazio relativo alla cartella che la contiene. Ad esempio, le invocazioni ricorsive che riguardano le entità `grades`, `homeworks` e `programs` vengono eseguite e portate a termine prima che venga calcolato lo spazio totale occupato dalla cartella `/user/rt/courses/cs016` che le contiene.

5.2 Analisi di algoritmi ricorsivi

Nel Capitolo 4 abbiamo presentato alcune tecniche matematiche per analizzare l'efficienza di un algoritmo, sulla base di una stima del numero di operazioni elementari che vengono eseguite dall'algoritmo stesso, e abbiamo usato la notazione O-grande per riassumere la relazione esistente tra il numero di operazioni e la dimensione del problema. In questo paragrafo vedremo come eseguire questo tipo di analisi del tempo d'esecuzione quando l'algoritmo è ricorsivo.

In un algoritmo ricorsivo, conteremo le singole operazioni che vengono eseguite durante *una specifica attivazione* del metodo che gestisce il flusso di controllo nel momento in cui viene eseguito. Detto in altro modo, per ogni invocazione del metodo, conteremo solamente il numero di operazioni che vengono eseguite all'interno del corpo di quella attivazione. Poi, potremo contare il numero complessivo di operazioni che vengono eseguite da un algoritmo ricorsivo sommando, per tutte le attivazioni, il numero di operazioni eseguite durante ciascuna singola attivazione (incidentalmente, questa è anche la procedura che usiamo per analizzare un metodo non ricorsivo che invoca un altro metodo all'interno del proprio corpo).

Per illustrare questa modalità di analisi, riprendiamo in esame i quattro algoritmi ricorsivi presentati nei Paragrafi che vanno dal 5.1.1 al 5.1.4: il calcolo del fattoriale, il disegno di un righello in pollici, la ricerca binaria e il calcolo dello spazio cumulativo occupato da una porzione di file system. In generale, per capire quante attivazioni ricorsive si verificano, possiamo basarci sull'intuizione, a sua volta basata sull'analisi di un diagramma di ricorsione; allo stesso modo, possiamo capire come i parametri possano influire, all'interno di ciascuna attivazione, sul numero di operazioni elementari eseguite all'interno del corpo del metodo. Ciò nonostante, ognuno di questi algoritmi ricorsivi ha una propria struttura e forma univoca.

Calcolare il fattoriale

L'analisi dell'efficienza del nostro metodo che calcola il fattoriale, descritto nel Paragrafo 5.1.1, è relativamente semplice. Nella Figura 5.1 abbiamo già visto un esempio di diagramma di ricorsione del nostro metodo `factorial`. Per calcolare $\text{factorial}(n)$, notiamo che servono complessivamente $n + 1$ attivazioni, perché il parametro diminuisce, passando da n nella prima invocazione a $n - 1$ nella seconda, e così via, finché non raggiunge il caso base quando il parametro vale zero.

Esaminando il corpo del metodo nel Codice 5.1, è altrettanto chiaro che ciascuna singola attivazione del metodo `factorial` esegue un numero costante di operazioni, quindi concludiamo che il numero complessivo di operazioni eseguite per il calcolo di $\text{factorial}(n)$ è $O(n)$, perché ci sono $n + 1$ attivazioni, ciascuna delle quali contribuisce al totale con $O(1)$ operazioni.

Disegnare un righello in pollici

Per analizzare l'efficienza dell'applicazione che disegna un righello in pollici, descritta nel Paragrafo 5.1.2, la domanda fondamentale a cui dobbiamo rispondere riguarda il numero complessivo di righe che vengono visualizzate per effetto dell'invocazione iniziale di `drawInterval(c)`, dove c indica la lunghezza centrale del righello. Questo è una misura ragionevole dell'efficienza complessiva dell'algoritmo, perché ogni riga visualizzata si basa su

un'invocazione del metodo ausiliario `drawLine` e ogni invocazione ricorsiva di `drawInterval` con un parametro diverso da zero effettua una e soltanto una invocazione diretta di `drawLine`.

L'esame del codice sorgente e del diagramma di ricorsione può essere d'aiuto per intuire il risultato: sappiamo che un'invocazione di `drawInterval(c)` con $c > 0$ fa partire due invocazioni di `drawInterval(c - 1)` e una singola invocazione di `drawLine`. Per dimostrare l'affermazione seguente ci avvarremo della precedente intuizione.

Proposizione 5.1: *Per qualsiasi $c \geq 0$, un'invocazione di `drawInterval(c)` visualizza esattamente $2^c - 1$ righe.*

Dimostrazione: Dimostriamo questa affermazione per *induzione* (si veda il Paragrafo 4.4.3). In effetti, l'induzione è una tecnica matematica che si rivela particolarmente naturale quando ci si trovi a dimostrare la correttezza e l'efficienza di un procedimento ricorsivo. Nel caso del disegno di un righello, notiamo che l'invocazione di `drawInterval(0)` non produce alcuna visualizzazione; inoltre, $2^0 - 1 = 1 - 1 = 0$. Questa osservazione sarà il caso base della nostra dimostrazione.

Più in generale, il numero di righe visualizzate da `drawInterval(c)` è uguale a un'unità più il doppio del numero di righe visualizzare da un'invocazione di `drawInterval(c - 1)`, perché fra tali due invocazioni ricorsive viene visualizzata la riga centrale. Per ipotesi induttiva, abbiamo quindi che il numero di righe visualizzate è proprio $1 + 2 \cdot (2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$. ■

Questa dimostrazione è un esempio di uno strumento matematico più rigoroso, noto come *equazione alle ricorrenze* o *equazione ricorrente*, che può essere utilizzato per analizzare il tempo d'esecuzione di un algoritmo ricorsivo e che sarà discusso in dettaglio nel Paragrafo 12.1.4, nel contesto degli algoritmi di ordinamento ricorsivi.

Effettuare una ricerca binaria

Parlando del tempo d'esecuzione dell'algoritmo di ricerca binaria, presentato nel Paragrafo 5.1.3, abbiamo osservato che durante ciascuna invocazione ricorsiva del metodo di ricerca binaria viene eseguito un numero costante di operazioni elementari. Quindi, il tempo d'esecuzione è proporzionale al numero di invocazioni ricorsive effettuate. Dimostreremo ora che durante una ricerca binaria all'interno di una sequenza di n elementi vengono eseguite al massimo $\lfloor \log n \rfloor + 1$ invocazioni ricorsive, dimostrando così l'affermazione seguente.

Proposizione 5.2: *L'algoritmo di ricerca binaria in un array ordinato di n elementi viene eseguito in un tempo $O(\log n)$.*

Dimostrazione: Per dimostrare questa affermazione, è decisivo il fatto che ad ogni invocazione ricorsiva il numero di elementi candidati, nei quali va effettuata la successiva ricerca, sia dato dal valore:

$$\text{high} - \text{low} + 1$$

Inoltre, il numero di candidati rimasti viene ridotto almeno alla metà del valore precedente da ogni invocazione ricorsiva. In particolare, dalla definizione di `mid`, risulta che il numero di candidati rimasti è:

$$(mid - 1) - low + 1 = \left\lfloor \frac{low + high}{2} \right\rfloor - low \leq \frac{high - low + 1}{2}$$

oppure:

$$high - (mid + 1) + 1 = high - \left\lfloor \frac{low + high}{2} \right\rfloor \leq \frac{high - low + 1}{2}$$

Inizialmente il numero di candidati è n : dopo la prima invocazione del metodo di ricerca binaria è al massimo $n/2$, dopo la seconda invocazione è al massimo $n/4$, e così via. Generalizzando, dopo la j -esima invocazione del metodo di ricerca binaria, il numero di elementi candidati rimasti da esaminare è al massimo $n/2^j$. Nel caso peggiore (che è quello di una ricerca infruttuosa), la ricorsione termina quando non ci sono più elementi candidati. Quindi, il numero massimo di invocazioni ricorsive che vengono eseguite è uguale al minimo numero intero r tale che:

—

In altre parole (ricordando che, quando vale 2, evitiamo di indicare la base dei logaritmi), r è il minimo numero intero per il quale è vero che $r > \log n$. Quindi, abbiamo che:

$$r = \lfloor \log n \rfloor + 1$$

e questo risultato implica che la ricerca binaria viene eseguita in un tempo $O(\log n)$. ■

Calcolare l'occupazione di spazio sul disco

L'ultimo algoritmo ricorsivo che abbiamo esaminato nel Paragrafo 5.1 calcolava lo spazio complessivo occupato sul disco da una determinata porzione di un file system. Per caratterizzare la "dimensione del problema", per la nostra analisi, indichiamo con n il numero di entità (file o cartelle) della porzione del file system che sono coinvolte dall'esecuzione dell'algoritmo (ad esempio, nel caso della porzione rappresentata nella Figura 5.6, il numero di entità è $n = 19$).

Per caratterizzare il tempo complessivo necessario per portare a termine un'iniziale invocazione di `diskUsage`, dobbiamo analizzare il numero totale di invocazioni ricorsive che vengono eseguite, oltre al numero di operazioni elementari eseguite al loro interno.

Cominciamo dimostrando che il numero di invocazioni del nostro metodo è esattamente uguale a n , in particolare viene effettuata un'invocazione del metodo per ogni entità del file system interessata dall'elaborazione. Intuitivamente, questo avviene perché, osservando il ciclo `for` del Codice 5.5, l'invocazione di `diskUsage` avente come argomento una particolare entità e del file system si verifica soltanto nel momento in cui viene elaborata l'unica entità di tipo cartella che contiene e , cosa che avviene una sola volta.

Per rendere più formale questa dimostrazione, possiamo definire il *livello di annidamento* o *profondità* (*nesting level*) di ciascun entità in modo che l'entità con cui iniziamo l'elaborazione abbia livello 0, le entità memorizzate direttamente in essa abbiano livello 1, le entità memorizzate all'interno di entità di livello 1 abbiano livello 2, e così via. Dimostreremo

ora, per induzione, che per ogni entità di livello k viene effettuata una e una sola invocazione ricorsiva di `diskUsage`. Come caso base, quando $k = 0$, l'unica invocazione effettuata è quella iniziale. Come passo induttivo, ipotizzando che ci sia una e una sola invocazione ricorsiva per ogni entità di livello k , possiamo affermare che viene effettuata una e una sola invocazione per ogni entità e di livello $k + 1$, che avviene all'interno del ciclo eseguito per l'entità di livello k che contiene e .

Avendo stabilito che avviene una sola invocazione ricorsiva per ogni entità del file system, torniamo alla domanda relativa al tempo d'esecuzione complessivo dell'algoritmo. Sarebbe bello se potessimo affermare che viene speso un tempo $O(1)$ per ogni singola invocazione del metodo, ma non è vero. Nonostante sia costante il numero di operazioni elementari dovute all'invocazione di `root.length()` per calcolare lo spazio occupato direttamente dall'entità in esame, quando l'entità è una cartella il corpo del metodo `diskUsage` esegue, poi, un ciclo `for`, che prevede un'iterazione per ogni entità contenuta all'interno di quella cartella. Nel caso peggiore, è possibile che una sola cartella contenga tutte le altre $n - 1$ entità.

Sulla base di questo ragionamento, potremmo concludere che vengono eseguite $O(n)$ invocazioni ricorsive, ognuna delle quali viene eseguita, nel caso peggiore, in un tempo $O(n)$, dando luogo a un tempo d'esecuzione complessivo che è $O(n^2)$. Anche se questo limite superiore è tecnicamente vero, non è un limite superiore molto stretto. Infatti, si può dimostrare la validità di un limite molto più stretto, secondo il quale l'algoritmo ricorsivo implementato da `diskUsage` termina in un tempo $O(n)$. Il limite più debole è pessimistico, perché ipotizza che il numero di entità che costituisce il caso peggiore per una cartella si verifichi per tutte le cartelle. Sebbene, però, sia possibile che alcune cartelle contengano un numero di entità proporzionale a n , questo non può accadere per tutte. Per dimostrare l'affermazione più stringente che abbiamo fatto, cerchiamo di calcolare il numero complessivo di iterazioni di tutti i cicli `for` che vengono eseguiti dalle varie invocazioni ricorsive. Affermiamo che questo numero è $n - 1$: ci basiamo sul fatto che quel ciclo effettua un'invocazione ricorsiva di `diskUsage` e abbiamo già concluso che il numero totale di tali invocazioni è n (compresa quella iniziale). Possiamo, quindi, concludere che ci sono $O(n)$ invocazioni ricorsive, ognuna delle quali richiede un tempo $O(1)$ al di fuori del ciclo, e che il numero complessivo di operazioni dovute a tutte le esecuzioni del ciclo è $O(n)$. Sommando questi tempi, si ottiene un andamento dell'algoritmo che è $O(n)$.

La dimostrazione che abbiamo fatto è più sofisticata di quella relativa ai precedenti esempi di ricorsione. L'idea su cui si basa è che a volte per una serie di operazioni si può ottenere un limite superiore più stretto considerandone l'effetto cumulativo, piuttosto che ipotizzare che ciascuna di esse possa raggiungere il proprio caso peggiore: un ragionamento che prende il nome di *analisi ammortizzata* e di cui vedremo un altro esempio nel Paragrafo 7.2.3 Inoltre, un file system è un esempio implicito di una struttura dati che è nota con il nome di *albero (tree)* e il nostro algoritmo che calcola lo spazio occupato sul disco è, in effetti, un esempio specifico di un algoritmo più generale: l'*attraversamento di un albero (tree traversal)*. Gli altri saranno argomento del Capitolo 8 e la nostra dimostrazione, che ha portato a concludere che il tempo d'esecuzione dell'algoritmo che calcola l'occupazione sul disco è $O(n)$, verrà generalizzata nel Paragrafo 8.4 per tutti gli attraversamenti.

5.3 Ulteriori esempi di ricorsione

In questo paragrafo vedremo altri esempi di utilizzo della ricorsione. Abbiamo organizzato il materiale sulla base del massimo numero di invocazioni ricorsive che possono essere eseguite a partire dal corpo di una singola attivazione.

- Se un'invocazione ricorsiva ne fa partire al massimo un'altra, parliamo di **ricorsione lineare**.
- Se un'invocazione ricorsiva ne fa partire al massimo altre due, parliamo di **ricorsione binaria o doppia**.
- Se un'invocazione ricorsiva ne fa partire più di altre due, parliamo di **ricorsione multipla**.

5.3.1 Ricorsione lineare

Se un metodo ricorsivo è progettato in modo che ogni esecuzione del suo corpo faccia iniziare al massimo una sola nuova invocazione ricorsiva, parliamo di **ricorsione lineare**. Tra le ricorsioni che abbiamo visto finora, l'implementazione del metodo che calcola il fattoriale (Paragrafo 5.1.1) è un chiaro esempio di ricorsione lineare. È interessante notare come anche l'algoritmo di ricerca binaria (Paragrafo 5.1.3) sia un esempio di **ricorsione lineare**, nonostante il termine "binario" che compare nel suo nome. Il codice della ricerca binaria (Codice 5.3) contiene un enunciato condizionale, con due diramazioni che portano a una ulteriore invocazione ricorsiva, ma, durante un'esecuzione del corpo del metodo, soltanto una delle diramazioni può essere seguita.

Una conseguenza della definizione di ricorsione lineare è che il relativo diagramma di ricorsione sarà costituito da un'unica sequenza di invocazioni, come abbiamo visto per il metodo che calcola il fattoriale, nella Figura 5.1 del Paragrafo 5.1.1. Il termine **ricorsione lineare** fa riferimento, appunto, alla struttura del diagramma di ricorsione, non all'analisi asintotica del tempo d'esecuzione: ad esempio, abbiamo già visto che la ricerca binaria, pur essendo una ricorsione lineare, viene eseguita in un tempo $O(\log n)$.

Sommare ricorsivamente gli elementi di un array

La ricorsione lineare può rivelarsi uno strumento utile per l'elaborazione di una sequenza, come un array in Java. Supponiamo, ad esempio, di voler calcolare la somma degli n numeri interi contenuti in un array. Possiamo risolvere questo problema usando la ricorsione lineare, osservando che: se $n = 0$ la somma è banalmente 0, altrimenti la somma è uguale alla somma dei primi $n - 1$ elementi dell'array più il suo ultimo elemento, come si può vedere nella Figura 5.9.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7

Figura 5.9: Calcolo ricorsivo della somma degli elementi di una sequenza, aggiungendo il suo ultimo elemento alla somma dei primi $n - 1$ elementi.

Il Codice 5.6 riporta un'implementazione dell'algoritmo ricorsivo per il calcolo della somma degli elementi di un array di numeri interi, basato sull'intuizione appena esposta.

Codice 5.6: Somma di un array di numeri interi usando la ricorsione lineare.

```

1  /** Restituisce la somma dei primi n numeri interi dell'array dato. */
2  public static int linearSum(int[] data, int n) {
3      if (n == 0)
4          return 0;
5      else
6          return linearSum(data, n-1) + data[n-1];
7  }

```

La Figura 5.10 riporta un piccolo esempio di diagramma di ricorsione del metodo `linearSum`. Per risolvere un problema di dimensione n , l'algoritmo `linearSum` effettua $n + 1$ invocazioni ricorsive, quindi la sua esecuzione richiederà un tempo $O(n)$, perché impiega un tempo costante per l'esecuzione della parte non ricorsiva di ciascuna invocazione. Inoltre, possiamo anche osservare che lo spazio di memoria utilizzato dall'algoritmo (oltre all'array che contiene i dati da elaborare) è, analogamente, $O(n)$, perché, nel momento in cui viene eseguita l'ultima invocazione ricorsiva (con $n = 0$), in memoria viene occupato uno spazio costante per ognuno dei dati di attivazione (*activation frame*), che sono $n + 1$.

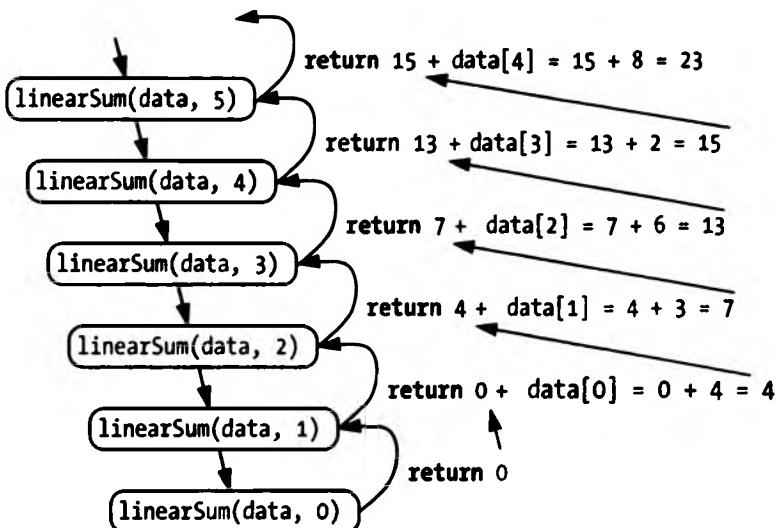


Figura 5.10: Diagramma di ricorsione per l'esecuzione di `linearSum(data, 5)`, con `data = 4,3,6,2,8`.

Invertire una sequenza usando la ricorsione

Prendiamo ora in esame il problema di invertire il contenuto di un array di n elementi, in modo che il suo primo elemento diventi l'ultimo, il secondo diventi il penultimo, e così via. Possiamo, di nuovo, risolvere questo problema usando la ricorsione lineare, osservando che l'inverso di una sequenza si può ottenere scambiando il primo e l'ultimo elemento, per poi invertire ricorsivamente la sequenza costituita da tutti gli altri elementi. Nel Codice 5.7

presentiamo un'implementazione di questo algoritmo; il metodo va inizialmente invocato come `reverseArray(data, 0, n-1)`.

Codice 5.7: Inverte gli elementi di un array usando la ricorsione lineare.

```

1  /** Inverte il contenuto del sottoarray data[low]...data[high], estremi compresi. */
2  public static void reverseArray(int[] data, int low, int high) {
3      if (low < high) {           // se il sottoarray ha almeno due elementi
4          int temp = data[low];   // scambia data[low] con data[high]
5          data[low] = data[high];
6          data[high] = temp;
7          reverseArray(data, low + 1, high - 1); // ricorsione sulla parte restante
8      }
9  }
```

Osserviamo che, ogni volta che si effettua un'invocazione ricorsiva, la porzione di array in esame conterrà due elementi in meno, come si può vedere nella Figura 5.11. Prima o poi, quindi, verrà raggiunto un caso base, quando la condizione `low < high` fallirà, perché `low == high`, se n è dispari, oppure perché `low == high + 1`, se n è pari.

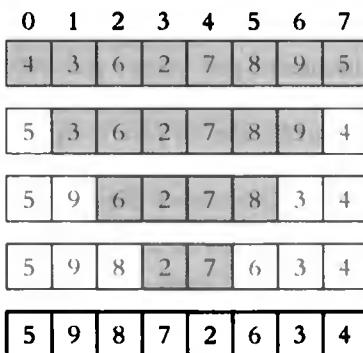


Figura 5.11: Schema d'esecuzione della ricorsione che inverte il contenuto di un array. Le porzioni evidenziate non sono ancora state invertite.

L'argomentazione precedente implica che l'algoritmo ricorsivo riportato nel Codice 5.7 terminerà certamente dopo $1 + \lfloor n/2 \rfloor$ invocazioni ricorsive. Dato che ogni invocazione viene eseguita in un tempo costante, l'intero processo di inversione della sequenza avviene in un tempo $O(n)$.

Algoritmi ricorsivi per l'elevamento a potenza

Come altro interessante esempio di utilizzo della ricorsione lineare, consideriamo il problema dell'elevamento a potenza, usando come base un numero x qualsiasi e come esponente un numero intero n non negativo, arbitrario. Vogliamo, cioè, calcolare la *funzione potenza* (*power function*), definita come $\text{power}(x, n) = x^n$ (in questa discussione usiamo il termine "potenza", *power*, tenendolo distinto dal metodo *pow* della classe *Math*, che fornisce la stessa funzionalità). Prenderemo in esame due diverse formulazioni ricorsive del problema, che porteranno alla progettazione di algoritmi aventi prestazioni molto diverse.

Una definizione ricorsiva banale deriva dal fatto che $x^n = x \cdot x^{n-1}$ per $n > 0$.

$$\text{power}(x, n) = \begin{cases} 1 & \text{se } n = 0 \\ x \cdot \text{power}(x, n - 1) & \text{altrimenti} \end{cases}$$

Questa definizione porta all'algoritmo ricorsivo riportato nel Codice 5.8.

Codice 5.8: Calcola la funzione potenza usando una ricorsione banale.

```

1  /** Calcola x elevato alla n-esima potenza, con n intero non negativo. */
2  public static double power(double x, int n) {
3      if (n == 0)
4          return 1;
5      else
6          return x * power(x, n-1);
7 }
```

Un'invocazione ricorsiva di questa versione di $\text{power}(x, n)$ viene eseguita in un tempo $O(n)$. Il suo diagramma di ricorsione ha una struttura molto simile a quello relativo alla funzione fattoriale, visto nella Figura 5.1, con il parametro che diminuisce di un'unità ad ogni invocazione e un lavoro costante svolto in ciascuno degli $n + 1$ livelli.

Tuttavia, esiste un modo molto più veloce per calcolare la funzione potenza, usando una definizione alternativa che impiega una tecnica di elevamento al quadrato. Poniamo $k = \lfloor n/2 \rfloor$, che in Java è equivalente a $k = n / 2$, se n è un valore di tipo `int`. Consideriamo l'espressione $(x^k)^2$. Quando n è pari, $\lfloor n/2 \rfloor = n/2$ e, quindi, $(x^k)^2 = (x^{n/2})^2 = x^n$. Quando n è dispari, invece, $\lfloor n/2 \rfloor = (n - 1)/2$ e $(x^k)^2 = x^{n-1}$, per cui $x^n = (x^k)^2 \cdot x$, proprio come $2^{13} = (2^6 \cdot 2^6) \cdot 2$. Questa analisi porta alla seguente definizione ricorsiva:

$$\text{power}(x, n) = \begin{cases} 1 & \text{se } n = 0 \\ (\text{power}(x, \lfloor \frac{n}{2} \rfloor))^2 \cdot x & \text{se } n > 0 \text{ è dispari} \\ (\text{power}(x, \lfloor \frac{n}{2} \rfloor))^2 & \text{se } n > 0 \text{ è pari} \end{cases}$$

Se implementassimo questa ricorsione effettuando *due* invocazioni ricorsive per calcolare $\text{power}(x, \lfloor n/2 \rfloor) \cdot \text{power}(x, \lfloor n/2 \rfloor)$, un diagramma di ricorsione riporterebbe un numero di invocazioni $O(n)$. Calcolando $\text{power}(x, \lfloor n/2 \rfloor)$ una sola volta e memorizzando il suo valore in una variabile come risultato parziale, per poi moltiplicarlo per se stesso, possiamo eseguire un numero di operazioni significativamente minore. Il Codice 5.9 riporta un'implementazione basata su questa definizione ricorsiva.

Codice 5.9: Calcola la funzione potenza usando ripetutamente l'elevamento al quadrato.

```

1  /** Calcola x elevato alla n-esima potenza, con n intero non negativo. */
2  public static double power(double x, int n) {
3      if (n == 0)
4          return 1;
5      else {
6          double partial = power(x, n/2); // sfrutta la divisione intera, che tronca
```

```

7   double result = partial * partial;
8   if (n % 2 == 1)           // se n è dispari, usa un ulteriore fattore x
9       result *= x;
10      return result;
11  }
12 }
```

Per aiutarci a visualizzare l'esecuzione del nostro algoritmo così migliorato, la Figura 5.12 mostra un diagramma di ricorsione per il calcolo di $\text{power}(2, 13)$.

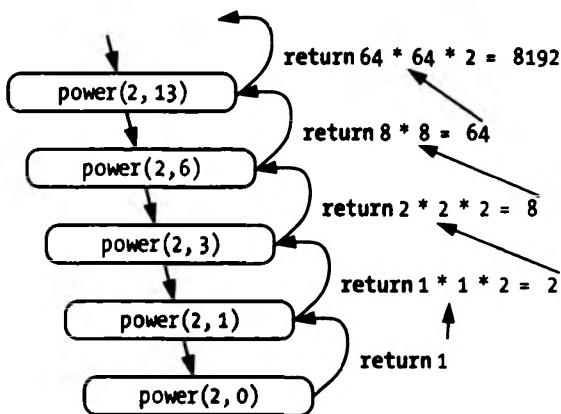


Figura 5.12: Diagramma di ricorsione per l'esecuzione di $\text{power}(2, 13)$.

Per analizzare il tempo d'esecuzione di questa seconda versione dell'algoritmo, osserviamo che in ciascuna invocazione del metodo $\text{power}(x, n)$ l'esponente è al massimo uguale alla metà del suo valore precedente. Come abbiamo visto nell'analisi della ricerca binaria, il numero di divisioni per 2 a cui si può sottoporre n prima di ottenere il valore 1 o un valore inferiore è $O(\log n)$. Quindi, la nostra nuova formulazione di power risolve il problema con $O(\log n)$ invocazioni ricorsive. Ogni singola attivazione del metodo usa un numero $O(1)$ di operazioni elementari (se si esclude l'invocazione ricorsiva), per cui il numero totale di operazioni richieste per calcolare $\text{power}(x, n)$ è $O(\log n)$: un miglioramento significativo rispetto all'algoritmo originale, che era $O(n)$.

La versione migliorata riduce in modo rilevante anche l'utilizzo della memoria. La prima versione ha una profondità di ricorsione $O(n)$ e, quindi, ha bisogno di memorizzare simultaneamente $O(n)$ dati di attivazione. Dato che la profondità di ricorsione della versione migliorata è $O(\log n)$, sarà $O(\log n)$ anche l'occupazione di memoria.

5.3.2 Ricorsione binaria o doppia

Quando un metodo effettua due invocazioni ricorsive, diciamo che usa una *ricorsione binaria* o *doppia*. Disegnando il righello in pollici, abbiamo già visto un esempio di ricorsione binaria (Paragrafo 5.1.2). Come ulteriore applicazione della ricorsione binaria, riprendiamo in esame il problema della somma degli n numeri interi contenuti in un array. Quando il numero di valori da sommare è zero o uno, il problema è banale. Con due o più valori,

possiamo calcolare ricorsivamente la somma della prima metà dei valori, poi la somma della seconda metà, per sommare, infine, questi due valori insieme. Il Codice 5.10 riporta la nostra prima implementazione di questo algoritmo, che viene inizialmente invocato come `binarySum(data, 0, n-1)`.

Codice 5.10: Somma gli elementi di una sequenza usando la ricorsione binaria.

```

1  /** Restituisce la somma del sottoarray data[low]...data[high], estremi compresi. */
2  public static int binarySum(int[] data, int low, int high) {
3      if (low > high)          // zero elementi nel sottoarray
4          return 0;
5      else if (low == high) {  // un elemento nel sottoarray
6          return data[low];
7      } else {
8          int mid = (low + high) / 2;
9          return binarySum(data, low, mid) + binarySum(data, mid+1, high);
10     }
11 }
```

Per analizzare l'algoritmo `binarySum`, consideriamo, per semplicità, il caso in cui n è una potenza di due. La Figura 5.13 mostra il diagramma di ricorsione relativo all'esecuzione dell'invocazione `binarySum(data, 0, 7)`. In ogni casella abbiamo indicato i valori dei parametri `low` e `high` per quell'invocazione. La dimensione dell'intervallo viene divisa a metà in conseguenza di ogni invocazione ricorsiva, per cui la profondità della ricorsione è $1 + \log_2 n$. Quindi, `binarySum` utilizza una quantità addizionale $O(\log n)$ di spazio di memoria, un deciso miglioramento rispetto allo spazio $O(n)$ usato dal metodo `linearSum` visto nel Codice 5.6. Tuttavia, il tempo d'esecuzione di `binarySum` è di nuovo $O(n)$, perché ci sono $2n - 1$ invocazioni del metodo, ciascuna delle quali richiede un tempo costante.

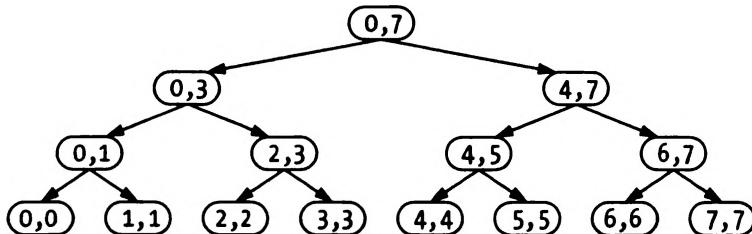


Figura 5.13: Diagramma di ricorsione per l'esecuzione di `binarySum(data, 0, 7)`.

5.3.3 Ricorsione multipla

Generalizzando la ricorsione binaria, definiamo la *ricorsione multipla* come un processo nel quale un metodo può attivare più di due invocazioni ricorsive. La ricorsione che abbiamo utilizzato per calcolare lo spazio occupato sul disco da una porzione di file system (come visto nel Paragrafo 5.1.4) è un esempio di ricorsione multipla, perché il numero di invocazioni ricorsive attivate durante una singola invocazione del metodo è uguale al numero di entità presenti all'interno di una determinata cartella del file system.

Un'altra tipica applicazione della ricorsione multipla riguarda l'enumerazione delle varie configurazioni possibili durante la soluzione di un rompicapo combinatorio. Ad esempio, questi sono esemplari del cosiddetto *rompicapo della somma* (*summation puzzle*):

$$\text{pot} + \text{pan} = \text{bib}$$

$$\text{dog} + \text{cat} = \text{pig}$$

$$\text{boy} + \text{girl} = \text{baby}$$

Per risolvere uno di questi rompicapi, occorre assegnare una diversa cifra numerica (cioè 0, 1, ..., 9) a ciascuna lettera presente nell'equazione, in modo da renderla vera. Solitamente risolviamo questi rompicapi usando il nostro spirito d'osservazione applicato a quel particolare problema, cercando di eliminare una dopo l'altra quelle configurazioni (cioè assegnamenti parziali di cifre a lettere) che non rispettano i vincoli, fino a ottenere un ridotto insieme di configurazioni possibili, che poi verifichiamo a mano.

Se il numero di configurazioni possibili non è troppo elevato, possiamo anche utilizzare un computer per enumerare banalmente tutte le possibilità, verificando la correttezza di ciascuna di esse, senza utilizzare in alcun modo il nostro spirito di osservazione. Un tale algoritmo può utilizzare la ricorsione multipla per analizzare in modo sistematico tutte le possibili configurazioni. Al fine di dare una descrizione sufficientemente generica da poter essere adattata ad altri rompicapi, consideriamo un algoritmo che enumera e verifichi tutte le sequenze di simboli di lunghezza k , senza ripetizioni, scelti in un universo U . Il Codice 5.11 riporta l'algoritmo che costruisce una sequenza di k elementi seguendo queste fasi:

1. Genera ricorsivamente le sequenze di $k - 1$ elementi
2. Aggiunge al termine di ciascuna di tali sequenze un elemento che non vi sia già presente

Durante l'esecuzione dell'algoritmo, usiamo l'insieme U per tenere traccia degli elementi non contenuti nella sequenza che si sta generando, in modo che un elemento e non sia ancora stato utilizzato se e solo se e appartiene a U .

Si può vedere il Codice 5.11 anche come un algoritmo che enumera tutti i possibili sottoinsiemi di U ordinati di dimensione k , verificando se ciascun sottoinsieme possa essere una soluzione del rompicapo.

Nel caso del rompicapo della somma, l'insieme U è $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ e ciascuna posizione nella sequenza corrisponde a una delle lettere date. Ad esempio, la prima posizione potrebbe essere la b , la seconda la o , la terza la y , e così via.

Codice 5.11: Algoritmo che risolve un rompicapo combinatorio enumerando e verificando tutte le possibili configurazioni.

Algoritmo `PuzzleSolve(k, S, U)`:

Input: Un numero intero k , una sequenza S e un insieme U

Output: L'enumerazione di tutte le estensioni di S che la portano ad avere lunghezza k usando senza ripetizioni elementi appartenenti a U

```

for ogni  $e$  in  $U$  do
    Aggiungi  $e$  alla fine di  $S$ 
    Elimina  $e$  da  $U$                                 { ora  $e$  è in uso }
    if  $k == 1$  then
        Verifica se  $S$  è una configurazione che risolve il rompicapo
        if  $S$  risolve il rompicapo then
            Aggiungi  $S$  al risultato prodotto come output      { una soluzione }
    else
        PuzzleSolve( $k - 1, S, U$ )                      { invocazione ricorsiva }
        Elimina  $e$  dalla fine di  $S$ 
        Reinserisci  $e$  in  $U$                             { ora  $e$  torna a essere utilizzabile }

```

La Figura 5.14 mostra il diagramma di ricorsione dell'invocazione $\text{PuzzleSolve}(3, S, U)$, dove S è vuota e $U = \{a, b, c\}$. Durante l'esecuzione, vengono generate e verificate tutte le permutazioni dei tre caratteri. Si noti che l'invocazione iniziale dà luogo a tre invocazioni ricorsive, ciascuna delle quali, a sua volta, ne genera altre due. Se avessimo eseguito $\text{PuzzleSolve}(3, S, U)$ su un insieme U contenente quattro elementi, l'invocazione iniziale avrebbe attivato quattro invocazioni ricorsive, ciascuna delle quali avrebbe generato un diagramma di ricorsione simile a quello riportato nella Figura 5.14.

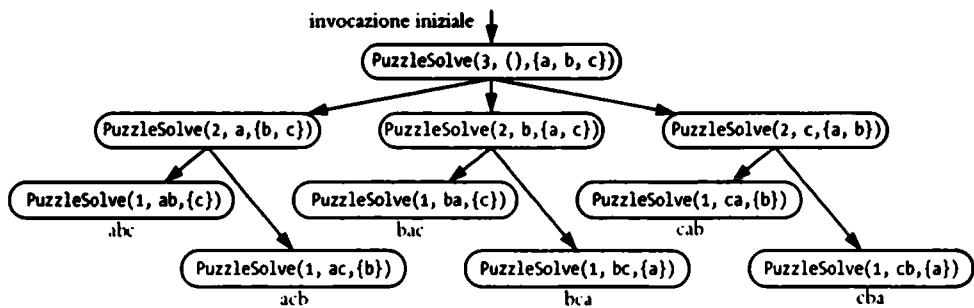


Figura 5.14: Diagramma di ricorsione per l'esecuzione di $\text{PuzzleSolve}(3, S, U)$, dove S è vuota e $U = \{a, b, c\}$. Questa esecuzione genera e verifica tutte le permutazioni dei caratteri a, b e c . Le permutazioni generate sono evidenziate al di sotto delle invocazioni relative.

5.4 Progettazione di algoritmi ricorsivi

Un algoritmo che usa la ricorsione ha solitamente questa forma:

- **Verifica dei casi base.** Iniziamo verificando un insieme di casi base (ce ne deve essere almeno uno), che devono essere definiti in modo che ogni possibile catena di invocazioni ricorsive arrivi a uno di essi; la gestione di ciascun caso base non deve usare la ricorsione.
- **Ricorsione.** Se non siamo in un caso base, effettuiamo una o più invocazioni ricorsive. Questo cosiddetto “passo ricorsivo” può comprendere un enunciato condizionale che decida quale ricorsione mettere in atto, scegliendo tra più alternative. Ogni possibile

invocazione ricorsiva deve essere definita in modo che faccia progredire il problema verso un caso base.

Aggiungere parametri a una ricorsione

Per progettare un algoritmo ricorsivo che risolva un determinato problema, è utile pensare ai diversi modi in cui si potrebbero definire suoi sottoproblemi che abbiano la stessa struttura generale del problema originario. Se si riscontrano difficoltà nell'individuazione della struttura ripetitiva che è necessaria per progettare un algoritmo ricorsivo, a volte è utile risolvere il problema in un ristretto numero di esempi, per vedere come, appunto, si potrebbero definire suoi sottoproblemi.

A volte per realizzare un buon progetto ricorsivo c'è bisogno di ridefinire il problema originario, in modo da rendere più agevole l'emersione di sottoproblemi simili. Spesso, a questo scopo, si rende parametrica la firma di un metodo. Ad esempio, eseguendo una ricerca binaria in un array, la firma più naturale del metodo, dal punto di vista dell'utilizzatore, sarebbe `binarySearch(data, target)`, mentre, nel Paragrafo 5.1.3, abbiamo definito il nostro metodo con la firma `binarySearch(data, target, low, high)`, usando i parametri aggiuntivi per delimitare i sottoarray durante il procedere della ricorsione. Questa modifica nel numero dei parametri è vitale per la ricerca binaria, così come altri esempi di questo capitolo (come `reverseArray`, `linearSum` e `binarySum`) hanno illustrato l'utilità della definizione di parametri aggiuntivi al fine di individuare sottoproblemi ricorsivi.

Se vogliamo dotare il nostro algoritmo di un'interfaccia pubblica più pulita, senza costringere l'utilizzatore a confrontarsi con i parametri aggiuntivi utili soltanto alla ricorsione, l'approccio più diffuso consiste nel rendere privato il metodo ricorsivo, introducendo un metodo pubblico che abbia soltanto i parametri che interessano all'utilizzatore e che, a sua volta, invochi il metodo privato con i parametri opportuni. Ad esempio, potremmo offrire, per un utilizzo pubblico, questa versione semplificata del metodo `binarySearch`:

```
/** Restituisce true se e solo se target appartiene all'array data. */
public static boolean binarySearch(int[] data, int target) {
    // invoca la versione con più parametri
    return binarySearch(data, target, 0, data.length - 1);
}
```

5.5 Ricorsioni fuori controllo

Anche se la ricorsione è uno strumento molto potente, è facile cadere in errore e usarla in malo modo. In questo paragrafo vedremo alcuni casi in cui una ricorsione implementata in modo scadente porta a una drastica inefficienza, analizzando anche alcune strategie per riconoscere ed evitare tali trappole.

Iniziamo da una rivisitazione del *problema dell'unicità degli elementi* visto nel Paragrafo 4.3.3: per determinare se tutti gli n elementi di una sequenza sono distinti possiamo usare una formulazione ricorsiva del problema. Come caso base, quando $n = 1$, gli elementi sono banalmente distinti. Per $n \geq 2$, gli elementi sono distinti se e solo se i primi $n - 1$ elementi sono distinti, gli ultimi $n - 1$ elementi sono distinti e il primo e l'ultimo elemento sono diversi (dato che quest'ultima è l'unica coppia di elementi che non viene già verificata nei sottoproblemi citati). Il Codice 5.12 riporta un'implementazione ricorsiva

basata su questa idea, il metodo `unique3` (per differenziarlo dai metodi `unique1` e `unique2` visti nel Capitolo 4).

Codice 5.12: Il metodo ricorsivo `unique3` che verifica se tutti gli elementi di una sequenza sono distinti.

```

1  /** Restituisce true se gli elementi in data[low]...data[high] sono distinti. */
2  public static boolean unique3(int[] data, int low, int high) {
3      if (low >= high) return true;                                // al massimo c'è un elemento
4      else if (!unique3(data, low, high-1)) return false; // duplicato nei primi n-1
5      else if (!unique3(data, low+1, high)) return false; // duplicato negli ultimi n-1
6      else return (data[low] != data[high]);           // primo e ultimo diversi?
7  }

```

Sfortunatamente, questo è un utilizzo della ricorsione terribilmente inefficiente. La parte non ricorsiva di ciascuna invocazione richiede un tempo $O(1)$, per cui il tempo d'esecuzione totale sarà proporzionale al numero totale di invocazioni ricorsive. Per analizzare il problema, indichiamo con n il numero di elementi in esame, cioè $n = \text{high} - \text{low}$.

Se $n = 1$ il tempo d'esecuzione di `unique3` è $O(1)$, perché in tal caso non viene effettuata alcuna invocazione ricorsiva. Nel caso generale, l'osservazione che ci guida è il fatto che una singola invocazione di `unique3` che debba risolvere un problema di dimensione n può dar luogo a due invocazioni ricorsive relative a problemi di dimensione $n - 1$. Queste due invocazioni di dimensione $n - 1$ potrebbero, a loro volta, generare quattro invocazioni (due ciascuna) relative a un problema di dimensione $n - 2$ e, di conseguenza, otto invocazioni di dimensione $n - 3$, e così via. Quindi, nel caso peggiore, il numero totale di invocazioni è dato dalla somma geometrica:

$$1 + 2 + 4 + \dots + 2^{n-1},$$

che, per la Proposizione 4.5, è uguale a $2^n - 1$. Ne consegue che il tempo d'esecuzione del metodo `unique3` è $O(2^n)$: si tratta di un metodo incredibilmente inefficiente per risolvere il problema dell'unicità degli elementi. La sua inefficienza non deriva dal fatto che usa la ricorsione, ma dal suo pessimo utilizzo, un problema che affronteremo nell'Esercizio C-5.12.

Una ricorsione inefficiente per calcolare i numeri di Fibonacci

Nel Paragrafo 2.2.3 abbiamo presentato una procedura che genera la sequenza dei numeri di Fibonacci, che può essere definita ricorsivamente in questo modo:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-2} + F_{n-1} \text{ per } n > 1.$$

Un'implementazione ricorsiva che si basi direttamente su questa definizione porta al metodo `fibonacciBad` presentato nel Codice 5.13, che calcola un numero di Fibonacci effettuando due invocazioni ricorsive ogni volta che non si trova nel caso base.

Codice 5.13: Calcola l' n -esimo numero di Fibonacci usando una ricorsione binaria.

```

1  /** Restituisce (in modo inefficiente) l'n-esimo numero di Fibonacci. */
2  public static long fibonacciBad(int n) {
3      if (n <= 1)
4          return n;
5      else
6          return fibonacciBad(n-2) + fibonacciBad(n-1);
7 }

```

Purtroppo questa implementazione diretta della formula di Fibonacci è un metodo terribilmente inefficiente. Il calcolo dell' n -esimo numero di Fibonacci eseguito in questo modo richiede un numero esponenziale di invocazioni del metodo. In particolare, se c_n è il numero di invocazioni effettuate durante l'esecuzione di `fibonacciBad(n)`, abbiamo:

$$c_0 = 1$$

$$c_1 = 1$$

$$c_2 = 1 + c_0 + c_1 = 1 + 1 + 1 = 3$$

$$c_3 = 1 + c_1 + c_2 = 1 + 1 + 3 = 5$$

$$c_4 = 1 + c_2 + c_3 = 1 + 3 + 5 = 9$$

$$c_5 = 1 + c_3 + c_4 = 1 + 5 + 9 = 15$$

$$c_6 = 1 + c_4 + c_5 = 1 + 9 + 15 = 25$$

$$c_7 = 1 + c_5 + c_6 = 1 + 15 + 25 = 41$$

$$c_8 = 1 + c_6 + c_7 = 1 + 25 + 41 = 67$$

Se proseguiamo lo schema di calcolo, vediamo che il numero di invocazioni più che raddoppia in corrispondenza di ogni coppia di indici consecutivi, cioè c_4 è maggiore del doppio di c_2 , c_5 è maggiore del doppio di c_3 , c_{56} è maggiore del doppio di c_{34} , e così via. Di conseguenza, $c_n > 2^{n/2}$, per cui l'invocazione `fibonacciBad(n)` genera un numero di invocazioni che risulta essere esponenziale in funzione di n .

Una ricorsione efficiente per calcolare i numeri di Fibonacci

Abbiamo ceduto alla tentazione di usare la formula ricorsiva, che ha portato a una pessima efficienza, per le modalità con cui l' n -esimo numero della sequenza di Fibonacci, F_n , dipende dai due valori precedenti, F_{n-2} e F_{n-1} , ma notiamo subito che, dopo aver calcolato F_{n-2} , l'invocazione che calcola F_{n-1} effettua una propria invocazione ricorsiva per calcolare ancora F_{n-2} , perché non sa che il valore di F_{n-2} è già stato calcolato in una fase precedente della ricorsione. È un lavoro ripetuto e, cosa ancora peggiore, entrambe queste invocazioni che calcolano F_{n-2} dovranno (ri)calcolare il valore di F_{n-3} , come farà l'invocazione che calcola F_{n-1} . Questo effetto di palleggio porta al tempo d'esecuzione esponenziale che caratterizza il metodo `fibonacciBad`.

Possiamo calcolare F_n in modo molto più efficiente usando una ricorsione in cui ciascuna invocazione del metodo effettua una sola invocazione ricorsiva. Per fare questo dobbiamo ridefinire l'obiettivo del metodo: invece di progettare un metodo che restituisce un unico

valore, che è l' n -esimo numero della sequenza di Fibonacci, definiamo un diverso metodo ricorsivo che restituisce un array contenente due numeri di Fibonacci consecutivi, { F_n , F_{n-1} }, definendo convenzionalmente $F_{-1} = 0$. Anche se restituire due numeri invece di uno solo sembra uno sforzo rilevante, questo trasferimento addizionale di informazione da un livello all'altro della ricorsione rende molto più semplice la prosecuzione della procedura di calcolo (in effetti, consente di evitare completamente la necessità di ricalcolare il secondo valore, che è già noto all'interno della ricorsione). Il Codice 5.14 mostra un'implementazione basata su questa strategia.

Codice 5.14: Calcola l' n -esimo numero di Fibonacci usando una ricorsione lineare.

```

1  /** Restituisce un array contenente due numeri di Fibonacci, F(n) e F(n-1). */
2  public static long[] fibonacciGood(int n) {
3      if (n <= 1) {
4          long[] answer = {n, 0};
5          return answer;
6      } else {
7          long[] temp = fibonacciGood(n - 1);           // restituisce {F(n-1),F(n-2)}
8          long[] answer = {temp[0] + temp[1], temp[0]}; // ci serve {F(n),F(n-1)}
9          return answer;
10     }
11 }
```

In termini di efficienza, la differenza tra le due ricorsioni che risolvono questo problema equivale alla differenza tra il giorno e la notte. Il metodo `fibonacciBad` impiega un tempo esponenziale, mentre affermiamo che il tempo d'esecuzione del metodo `fibonacciGood` è $O(n)$. Infatti, ogni invocazione ricorsiva di `fibonacciGood` diminuisce il proprio argomento n di un'unità, quindi il relativo diagramma di ricorsione contiene una serie di n invocazioni del metodo. Dato che il lavoro svolto dalla parte non ricorsiva di ciascuna invocazione richiede un tempo costante, l'intera elaborazione viene svolta in un tempo $O(n)$.

5.5.1 Massima profondità di ricorsione in Java

Un altro pericolo derivante da un cattivo uso della ricorsione è noto come *ricorsione infinita*. Se ogni invocazione ricorsiva effettua un'altra invocazione ricorsiva, senza mai raggiungere un caso base, avremo una sequenza infinita di tali invocazioni: è un errore disastroso. Una ricorsione infinita può consumare rapidamente le risorse di calcolo, non soltanto per il suo repentino utilizzo della CPU, ma anche perché ogni invocazione ricorsiva crea i propri dati di attivazione, che richiedono ulteriore memoria. Un esempio di ricorsione mal progettata che usa risorse senza alcun ritegno è il seguente:

```

/** Versione con ricorsione infinita, da NON invocare. */
public static int fibonacci(int n) {
    return fibonacci(n); // in fin dei conti, F(n) è uguale a F(n)
}
```

Ci possono, però, essere errori molto più subdoli che danno origine a una ricorsione infinita. Tornando alla nostra implementazione della ricerca binaria (nel Codice 5.3), quando effettuiamo l'invocazione ricorsiva relativa alla porzione destra della sequenza (alla riga 15),

specifichiamo che il sottoarray da esaminare va dall'indice `mid+1` all'indice `high`. Se, invece, quella riga fosse stata scritta così:

```
return binarySearch(data, target, mid, high); // mid invece di mid+1
```

si potrebbe verificare una ricorsione infinita. In particolare, quando si cerca in un sottoarray di due elementi, è possibile che l'invocazione ricorsiva si riferisca allo stesso sottoarray.

Ogni programmatore dovrebbe fare molta attenzione al fatto che ciascuna invocazione ricorsiva progredisca, in qualche modo, verso un caso base (ad esempio, facendo decrescere il valore di un parametro a ogni invocazione). Per contrastare il rischio di ricorsione infinita, i progettisti del linguaggio Java hanno intenzionalmente limitato lo spazio complessivo utilizzabile per memorizzare i dati di attivazione per le invocazioni di metodi simultaneamente attive. Se viene raggiunto tale limite, la Java Virtual Machine lancia un'eccezione di tipo `StackOverflowError` (nel Paragrafo 6.1 ci occuperemo della struttura dati "stack" citata in questo nome). Il valore esatto di questo limite dipende alla particolare versione di Java, ma tipicamente consente circa mille invocazioni simultanee.

Per molte applicazioni della ricorsione le mille invocazioni consentite sono sufficienti. Ad esempio, il nostro metodo `binarySearch` (visto nel Paragrafo 5.1.3) ha una profondità di ricorsione $O(\log n)$, quindi, perché tale limite di profondità venga raggiunto, l'array in cui si effettua la ricerca deve avere circa 2^{1000} elementi, un valore molto più elevato del numero di atomi che si stima siano presenti nell'universo. Tuttavia, abbiamo già visto numerose ricorsioni lineari, che hanno una profondità di ricorsione proporzionale a n : il limite sulla profondità di ricorsione imposto da Java potrebbe impedire il funzionamento di tali algoritmi.

È possibile riconfigurare la Java Virtual Machine in modo da incrementare lo spazio destinato ai dati di attivazione dei metodi, usando l'opzione `-Xss` al momento dell'esecuzione, fornendola come parametro sulla riga di comando o tramite un IDE. In alternativa, spesso è possibile analizzare in dettaglio un algoritmo ricorsivo, per implementarlo in modo diverso, usando direttamente cicli invece di invocazioni di metodi per realizzare il meccanismo di ripetizione desiderato. Per concludere il capitolo, parleremo proprio di questo approccio.

5.6 Eliminare la ricorsione in coda

Il vantaggio principale apportato dall'approccio ricorsivo alla progettazione di algoritmi è, in sintesi, dovuto al fatto che consente di sfruttare la struttura ripetitiva presente in molti problemi. Facendo in modo che la descrizione del nostro algoritmo utilizzi la struttura ripetitiva mediante una ricorsione, spesso siamo in grado di evitare l'analisi di situazioni complesse e la progettazione di cicli annidati. Questo approccio può portare a descrizioni di algoritmi di più facile lettura e comprensione, pur rimanendo abbastanza efficienti.

Questi vantaggi derivanti dalla ricorsione hanno, però, un costo, ancorché spesso modesto. In particolare, la Java Virtual Machine deve gestire i dati di attivazione che servono per tenere traccia dello stato di ciascuna invocazione annidata. Quando la memoria del calcolatore è preziosa, può essere utile dedurre un'implementazione non ricorsiva di algoritmi ricorsivi.

In generale, per convertire un algoritmo ricorsivo in uno non ricorsivo si può utilizzare la struttura dati che chiamiamo “stack” (cioè *pila*, nel senso di “una sequenza di oggetti impilati uno sopra l’altro”), e che vedremo nel Paragrafo 6.1, per gestire autonomamente l’annidamento della struttura, invece che lasciarlo fare all’interprete del linguaggio. Sebbene questa strategia non faccia altro che spostare l’occupazione di memoria dalla pila usata dall’interprete alla nostra pila, possiamo ottenere un’ulteriore riduzione della memoria utilizzata valutando con cura quali informazioni vadano effettivamente memorizzate.

Una situazione ancora più favorevole si ha, poi, nei casi di alcune forme di ricorsione, che possono essere eliminate senza l’uso di memoria ausiliaria. Una di tali forme è la *ricorsione in coda (tail recursion)*: una ricorsione si dice “in coda” se ogni invocazione ricorsiva effettuata in un determinato contesto è l’ultima azione compiuta in quel contesto, con il valore restituito dall’invocazione ricorsiva (se questa restituisce un valore) che viene a sua volta immediatamente restituito dall’invocazione circostante. Una ricorsione in coda deve necessariamente essere una ricorsione lineare, dal momento che non c’è la possibilità di effettuare una seconda invocazione ricorsiva se si deve restituire immediatamente il risultato della prima.

Tra i metodi ricorsivi illustrati in questo capitolo, il metodo `binarySearch` (visto nel Codice 5.3) e il metodo `reverseArray` (visto nel Codice 5.7) sono esempi di ricorsione in coda, mentre alcuni altri nostri esempi di ricorsione lineare assomigliano molto a una ricorsione in coda, ma tecnicamente non lo sono. Ad esempio, il nostro metodo `factorial`, visto nel Codice 5.1, *non* è una ricorsione in coda, perché si conclude con l’enunciato:

```
return n * factorial(n-1);
```

Questa non è una ricorsione in coda, perché dopo che l’invocazione ricorsiva è terminata viene eseguita un’ulteriore moltiplicazione, per calcolare il valore da restituire. Per motivi analoghi, non sono ricorsioni in coda il metodo `linearSum` (Codice 5.6), entrambi i metodi `power` (Codice 5.8 e 5.9) e il metodo `fibonacciGood` (Codice 5.13).

Le ricorsioni in coda sono in qualche modo speciali, perché si possono implementare automaticamente in modo non ricorsivo racchiudendone il corpo in un ciclo che gestisca la ripetizione e sostituendo un’invocazione ricorsiva con nuovi parametri con l’assegnazione di nuovi valori ai parametri esistenti. In effetti, molti linguaggi di programmazione sono in grado di convertire la ricorsione in coda in questo modo come forma di ottimizzazione del codice.

Come esempio concreto, nel Codice 5.15 il nostro metodo `binarySearch` è stato implementato in modo non ricorsivo. Subito prima dell’inizio del ciclo `while`, inizializziamo le variabili `low` e `high` in modo che rappresentino l’intera estensione dell’array. Poi, durante ciascuna iterazione del ciclo, troviamo il valore cercato oppure restriniamo l’estensione del sottoarray contenente i candidati. Nel punto in cui nella versione originale effettuavamo l’invocazione ricorsiva `binarySearch(data, target, low, mid - 1)`, ora scriviamo semplicemente `high = mid - 1` e, poi, proseguiamo con la successiva iterazione del ciclo. La condizione che nella versione ricorsiva esprimeva il caso base, cioè `low > high`, è stata semplicemente sostituita dalla condizione opposta che controlla il ciclo, `while (low <= high)`. In questa nuova implementazione, restituiamo infine `false` (per indicare il fallimento di una ricerca) nel momento in cui il ciclo termina senza aver mai restituito `true` al proprio interno.

Codice 5.15: Un'implementazione non ricorsiva della ricerca binaria.

```

1  /** Restituisce true se e solo target si trova nell'array data. */
2  public static boolean binarySearchIterative(int[] data, int target) {
3      int low = 0;
4      int high = data.length - 1;
5      while (low <= high) {
6          int mid = (low + high) / 2;
7          if (target == data[mid])
8              return true;           // valore trovato
9          else if (target < data[mid])
10             high = mid - 1;      // considera solo i valori a sinistra di mid
11         else
12             low = mid + 1;        // considera solo i valori a destra di mid
13     }
14     return false;            // ciclo terminato senza successo
15 }
```

La maggior parte delle altre ricorsioni lineari si possono esprimere in modo abbastanza efficiente con un'iterazione, anche se formalmente non sono delle ricorsioni in coda. Ad esempio, è possibile progettare banalmente delle implementazioni non ricorsive per calcolare il fattoriale, per calcolare i numeri di Fibonacci, per sommare gli elementi di un array o per invertire il contenuto di un array. Il Codice 5.16 riporta, come esempio, un metodo non ricorsivo che inverte il contenuto di un array (da confrontare con il precedente metodo ricorsivo, visto nel Codice 5.7).

Codice 5.16: Inverte gli elementi di un array usando un'iterazione.

```

1  /** Inverte il contenuto dell'array dato. */
2  public static void reverseArray(int[] data) {
3      int low = 0, high = data.length - 1;
4      while (low < high) {           // scambia data[low] con data[high]
5          int temp = data[low];
6          data[low++] = data[high]; // post-incremento di low
7          data[high--] = temp;    // post-decremento di high
8      }
9 }
```

5.7 Esercizi

Riepilogo e approfondimento

- R-5.1 Descrivere un algoritmo ricorsivo che trovi il massimo elemento presente in un array, A , contenente n elementi. Qual è il tempo d'esecuzione e l'occupazione di spazio?
- R-5.2 Spiegare come si possa modificare l'algoritmo ricorsivo di ricerca binaria in modo che restituiscia l'indice della cella dell'array in cui è stato trovato il valore cercato, oppure -1 se il valore cercato non è presente.
- R-5.3 Disegnare il diagramma di ricorsione per il calcolo di $\text{power}(2, 5)$, usando l'algoritmo tradizionale, implementato nel Codice 5.8.
- R-5.4 Disegnare il diagramma di ricorsione per il calcolo di $\text{power}(2, 18)$, usando l'algoritmo che eleva ripetutamente al quadrato, implementato nel Codice 5.9.

- R-5.5 Disegnare il diagramma di ricorsione per l'esecuzione del metodo `reverseArray(data, 0, 4)`, visto nel Codice 5.7, quando l'array `data` contiene i valori 4, 3, 6, 2, 6.
- R-5.6 Disegnare il diagramma di ricorsione per l'esecuzione del metodo `PuzzleSolve(3, S, U)`, visto nel Codice 5.11, quando S è vuota e $U = \{a, b, c, d\}$.
- R-5.7 Descrivere un algoritmo ricorsivo per il calcolo dell' n -esimo *numero armonico*, definito come $H_n = \sum_{k=1}^n 1/k$.
- R-5.8 Descrivere un algoritmo ricorsivo per convertire una stringa di cifre nel numero intero che rappresenta. Ad esempio, '13531' rappresenta il numero intero 13531.
- R-5.9 Progettare un'implementazione non ricorsiva della versione del metodo `power` descritta nel Codice 5.9 che usa ripetutamente l'elevamento al quadrato.
- R-5.10 Descrivere una strategia di utilizzo della ricorsione per calcolare la somma di tutti gli elementi di un array bidimensionale $n \times n$ di numeri interi.

Creatività

- C-5.11 Descrivere un algoritmo ricorsivo che calcoli la parte intera del logaritmo in base 2 di n , usando soltanto addizioni e divisioni intere.
- C-5.12 Descrivere un algoritmo ricorsivo efficiente per risolvere il problema dell'unicità degli elementi, che, senza utilizzare l'ordinamento, venga eseguito al massimo in un tempo $O(n^2)$ nel caso peggiore.
- C-5.13 Descrivere un algoritmo ricorsivo che calcoli il prodotto di due numeri interi positivi, m e n , usando soltanto addizioni e sottrazioni.
- C-5.14 Nel Paragrafo 5.2 abbiamo dimostrato per induzione che il numero di *righe* visualizzate da un'invocazione di `drawInterval(c)` è $2^c - 1$. Un'altra domanda interessante riguarda il numero di *segmenti* ("dash") che vengono visualizzati durante tale procedura. Dimostrare, sempre per induzione, che tale numero è $2^{c+1} - c - 2$.
- C-5.15 Scrivere un metodo ricorsivo che visualizzi tutti i sottoinsiemi di un insieme di n elementi (senza ripetere nessun sottoinsieme).
- C-5.16 Nel rompicapo delle *Torri di Hanoi*, disponiamo di una piattaforma con tre pioli (*peg*), a , b e c , verticali. Sul piolo a sono impilati n dischi, ciascuno più piccolo di quello sottostante, in modo che il più piccolo si trovi in cima e il più grande, ovviamente, alla base della pila. Il rompicapo consiste nel trasferire tutti i dischi dal piolo a al piolo c , spostando un disco per volta, in modo che non si verifichi mai la situazione in cui un disco è impilato sopra a uno di dimensioni inferiori. La Figura 5.15 mostra un esempio nel caso $n = 4$. Descrivere un algoritmo ricorsivo che risolva il rompicapo delle Torri di Hanoi per n arbitrario. Suggerimento: si consideri prima il sottoproblema relativo allo spostamento di tutti i dischi tranne l' n -esimo, dal piolo a a un altro piolo, usando il terzo piolo come "spazio di memorizzazione temporaneo".
- C-5.17 Scrivere un breve metodo ricorsivo in Java che riceva una stringa di caratteri, s , e visualizzi la stringa inversa. Ad esempio, la stringa inversa di "pots&pans" è "snap&stop".
- C-5.18 Scrivere un breve metodo ricorsivo in Java che determini se una stringa s è un palindromo, cioè è uguale alla propria stringa inversa. Esempi di palindromo sono "racecar" e "gohangasalamimalasagnahog".

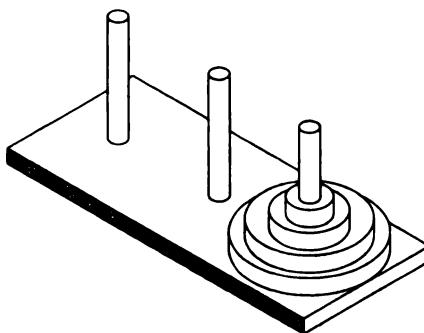


Figura 5.15: Una raffigurazione del rompicapo delle Torri di Hanoi.

- C-5.19 Usare la ricorsione per scrivere un metodo Java che determini se la stringa s ha più vocali che consonanti.
- C-5.20 Scrivere un breve metodo ricorsivo in Java che alteri la sequenza degli elementi di un array di numeri interi in modo che tutti i valori pari compaiano prima di tutti i valori dispari.
- C-5.21 Dato un array A di numeri interi, non ordinato, e un numero intero k , descrivere un algoritmo ricorsivo per alterare la sequenza degli elementi di A in modo che tutti gli elementi non maggiori di k vengano prima di tutti gli elementi maggiori di k . Qual è il tempo d'esecuzione dell'algoritmo applicato a un array di n elementi?
- C-5.22 Dato un array A contenente n numeri interi distinti, ordinati in senso crescente, e un numero k , descrivere un algoritmo ricorsivo che trovi, se esistono, due numeri interi appartenenti all'array A tali che la loro somma sia k . Qual è il tempo d'esecuzione dell'algoritmo?
- C-5.23 Descrivere un algoritmo ricorsivo che verifichi se un array A di numeri interi contiene un numero $A[i]$ che sia la somma di due numeri che compaiono in posizioni precedenti di A , cioè tale che sia $A[i] = A[j] + A[k]$, con $j, k < i$.
- C-5.24 Isabel ha scoperto un modo interessante per sommare i valori presenti in un array A contenente n numeri interi, con n che sia una potenza di due. Si crea un array B di dimensione pari alla metà della dimensione di A e si pone $B[i] = A[2i] + A[2i + 1]$, per $i = 0, 1, \dots, (n/2) - 1$. Se B ha dimensione 1, il risultato prodotto è $B[0]$, altrimenti si sostituisce A con B e si ripete la procedura. Qual è il tempo d'esecuzione dell'algoritmo?
- C-5.25 Descrivere un algoritmo ricorsivo veloce che inverta il contenuto di una lista semplicemente concatenata L , in modo che l'ordinamento tra i nodi risulti invertito rispetto alla situazione iniziale.
- C-5.26 Fornire una definizione ricorsiva di una classe che rappresenti una lista semplicemente concatenata senza fare uso di una classe per rappresentare i nodi.

Progettazione

- P-5.27 Implementare un metodo ricorsivo avente la firma `find(path, filename)` che produca un elenco di tutte le entità della porzione di file system avente radice nell'entità `path` e il cui nome sia `filename`.

- P-5.28 Scrivere un programma che risolva “rompicapi della somma” enumerando e verificando tutte le configurazioni possibili. Usando il programma così progettato, risolvere i tre rompicapi descritti nel Paragrafo 5.3.3.
- P-5.29 Fornire un’implementazione non ricorsiva del metodo `drawInterval` che visualizza parte di un righello in pollici come descritto nel Paragrafo 5.1.2. Se c è la lunghezza del segmento centrale, il metodo deve visualizzare esattamente $2^c - 1$ righe. Incrementando un contatore da 0 a $2^c - 2$, il numero di tratti di segmento visualizzati per ciascuna riga deve essere uguale al numero di cifre 1 consecutive presenti alla fine della rappresentazione binaria del contatore.
- P-5.30 Scrivere un programma che sia in grado di risolvere il problema delle Torri di Hanoi (descritto nell’Esercizio C-5.16).

Note

L’uso della ricorsione nei programmi fa parte della storia dell’informatica (si veda, ad esempio, l’articolo di Dijkstra [31]) e sta anche alla base dei linguaggi di programmazione funzionale (si veda, ad esempio, il libro di Abelson, Sussman e Sussman [1]). È forse interessante notare che la ricerca binaria fu pubblicata per la prima volta nel 1946, ma la sua prima formulazione pienamente corretta è del 1962. Per un’ulteriore analisi, si vedano i lavori di Bentley [13] e Lesuisse [64].

6

Pile, code e code doppie

6.1 Pile (*stack*)

Una pila (*stack*) è un contenitore di oggetti che vengono inseriti e rimossi sulla base di una politica di gestione denominata LIFO, cioè *last-in, first-out*: l'ultimo entrato sarà il primo a uscire. Un utilizzatore può inserire oggetti in una pila in qualsiasi momento, ma può ispezionare o rimuovere soltanto l'oggetto che è stato inserito più recentemente (cioè quello che si trova, come si dice, "in cima" alla pila, *top of the stack*). Il nome "pila" deriva dalla metafora di una pila di piatti all'interno di un apposito distributore a molla, come quelli usati nei ristoranti. In quel caso le operazioni fondamentali riguardano l'inserimento di un piatto sulla pila (ponendolo in cima alla pila e *spingendolo* verso il basso, premendo contro la molla, da cui il termine inglese "push", che significa proprio spingere) e l'estrazione del piatto che si trova in cima alla pila, spinto dalla molla fino al superamento di un sistema di blocco *a scatto* (da cui il termine inglese "pop", un verbo onomatopeico che significa "fare il botto"). Un esempio di pila forse ancora più simpatico è il contenitore/erogatore di caramelle PEZ→, che conserva caramelle alla menta in un tubo a molla che "spara" fuori (*pop*) quella più in cima alla pila ogni volta che il coperchio viene aperto (come si può vedere nella Figura 6.1).

Le pile sono strutture dati fondamentali e vengono utilizzate in molteplici applicazioni, tra le quali citiamo:

Esempio 6.1: I navigatori ("browser") web per Internet memorizzano in una pila gli indirizzi dei siti visitati più recentemente. Ogni volta che l'utente visita un nuovo sito, il suo indirizzo viene "spinto" in cima alla pila degli indirizzi. In questo modo il browser consente all'utente di "estrarre" i siti visitati in precedenza, a ritroso, usando il pulsante "back".

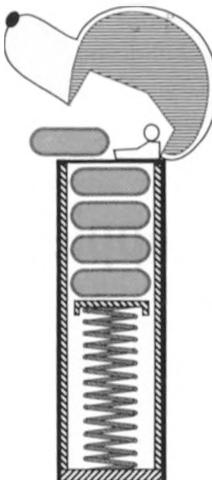


Figura 6.1: Un disegno schematico di un erogatore PEZ→ che implementa fisicamente il tipo di dato astratto "pila" o stack (PEZ→ è un marchio commerciale registrato da PEZ Candy, Inc.).

Esempio 6.2: Gli editor ("modificatori") di testo mettono solitamente a disposizione dell'utente un meccanismo di annullamento delle operazioni ("undo") che consente di annullare l'effetto delle operazioni di modifica più recenti, procedendo a ritroso, facendo tornare il documento alle situazioni precedenti. Tale operazione di annullamento viene resa possibile dalla conservazione in una pila delle azioni di modifica del testo.

6.1.1 La pila come tipo di dato astratto

Le pile sono le più semplici tra tutte le strutture dati, tuttavia sono anche tra le più importanti, perché vengono utilizzate in molte applicazioni, tra loro anche assai diverse, oltre che come componente interno di molte strutture dati e algoritmi complessi. Formalmente, una pila è un tipo di dato astratto (*abstract data type*, ADT) che consente l'utilizzo dei due metodi di aggiornamento seguenti:

- push(*e*):** Aggiunge l'elemento *e* in cima alla pila.
- pop():** Elimina dalla pila l'elemento che si trova in cima e lo restituisce (oppure restituisce `null` se la pila è vuota).

Inoltre, per comodità del programmatore, una pila solitamente mette a disposizione i seguenti metodi d'accesso:

- top():** Restituisce l'elemento che si trova in cima alla pila, senza eliminarlo (oppure restituisce `null` se la pila è vuota).
- size():** Restituisce il numero di elementi presenti nella pila.
- isEmpty():** Restituisce `true` se e solo se la pila è vuota.

Per convenzione, ipotizziamo che gli elementi che vengono inseriti in una pila possano essere di qualsiasi tipo e che ogni pila sia vuota nel momento in cui viene creata.

Esempio 6.3: La tabella seguente mostra una serie di operazioni eseguite su una pila S di numeri interi, inizialmente vuota, e gli effetti prodotti su di essa.

Operazione	Valore restituito	Contenuto della pila
push(5)	-	(5)
push(3)	-	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	-	(7)
push(9)	-	(7, 9)
top()	9	(7, 9)
push(4)	-	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	-	(7, 9, 6)
push(8)	-	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

La pila come interfaccia in Java

Per formalizzare la nostra astrazione di pila, definiamo la sua *interfaccia per la programmazione di applicazioni* (API, *application programming interface*) sotto forma di una *interfaccia Java*, che descrive i nomi dei metodi messi a disposizione dal tipo di dato astratto e il modo in cui essi vanno dichiarati e invocati. Questa interfaccia è definita nel Codice 6.1.

Codice 6.1: L'interfaccia Stack con i commenti secondo lo stile previsto da Javadoc (presentato nel Paragrafo 1.9.4). Si noti l'utilizzo del parametro di tipo, E , che consente di progettare una pila che possa contenere elementi di qualsiasi tipo (purché sia un riferimento).

```

1  /**
2   * Una raccolta di oggetti che vengono inseriti e eliminati secondo il principio
3   * last-in first-out. Anche se ha uno scopo simile, questa interfaccia è diversa da
4   * java.util.Stack.
5   *
6   * @author Michael T. Goodrich
7   * @author Roberto Tamassia
8   * @author Michael H. Goldwasser
9   */
10  public interface Stack<E> {
11
12      /**
13       * Restituisce il numero di elementi presenti nella pila.
14       * @return il numero di elementi presenti nella pila
15      */

```

```

16 int size();
17
18 /**
19 * Verifica se la pila è vuota.
20 * @return true se e solo se la pila è vuota
21 */
22 boolean isEmpty();
23
24 /**
25 * Inserisce un elemento in cima alla pila.
26 * @param e l'elemento da inserire
27 */
28 void push(E e);
29
30 /**
31 * Restituisce l'elemento in cima alla pila, senza eliminarlo.
32 * @return l'elemento in cima alla pila (o null se la pila è vuota)
33 */
34 E top();
35
36 /**
37 * Elimina e restituisce l'elemento che si trova in cima alla pila.
38 * @return l'elemento eliminato (o null se la pila è vuota)
39 */
40 E pop();
41 }

```

Per questa definizione ci siamo basati sull'*infrastruttura di programmazione generica (generics framework)* di Java, descritta nel Paragrafo 2.5.2, che consente agli elementi memorizzati nella pila di essere esemplari di qualsiasi tipo di classe, indicata con `<E>`. Ad esempio, una variabile che debba rappresentare una pila di numeri interi può essere dichiarata di tipo `Stack<Integer>`. Il parametro formale di tipo, `E`, viene quindi utilizzato come tipo del parametro ricevuto dal metodo `push`, così come tipo del valore restituito tanto da `pop` quanto da `top`.

Ricordiamo, dalla discussione sulle interfacce che è stata condotta nel Paragrafo 2.3.1, che queste servono come definizioni di tipi di dati ma non se ne possono creare esemplari in modo diretto. Perché un tale tipo di dato astratto sia di una qualche utilità, dobbiamo progettare una o più classi concrete che implementino i metodi dell'interfaccia corrispondente all'ADT. Nelle pagine seguenti vedremo due di tali implementazioni dell'interfaccia `Stack`: una che memorizza gli elementi in un array e un'altra che usa una lista concatenata.

La classe `java.util.Stack`

Vista l'importanza della pila come ADT, nella libreria di Java è presente, sin dalla sua prima versione, una classe concreta di nome `java.util.Stack`, che realizza la semantica LIFO caratteristica della pila. Tuttavia, questa classe `Stack` di Java rimane nella libreria soltanto per motivi storici, di compatibilità con il passato, e la sua interfaccia non è più coerente con quella della maggior parte delle altre strutture dati presenti nella libreria. In effetti, l'attuale documentazione di tale classe `Stack` raccomanda che non venga utilizzata, perché la funzionalità LIFO (oltre ad altre) è realizzata da una struttura dati più generale che prende il nome di "coda doppia" (*double-ended queue*), di cui parleremo nel Paragrafo 6.3.

A titolo di confronto, la Tabella 6.1 propone una corrispondenza uno a uno tra l'interfaccia da noi definita per il tipo di dato astratto pila e la classe `java.util.Stack`. Oltre

ad alcune differenze nei nomi dei metodi, osserviamo che i metodi `pop` e `peek` della classe `java.util.Stack` lanciano una propria eccezione, `EmptyStackException`, nel caso in cui vengano invocati quando la pila è vuota (mentre nella nostra astrazione viene restituito `null`).

Tabella 6.1: I metodi del nostro ADT pila e i corrispondenti metodi della classe `java.util.Stack`, con le differenze evidenziate a lato.

Nostro ADT pila	Classe <code>java.util.Stack</code>
<code>size()</code>	<code>size()</code>
<code>isEmpty()</code>	<code>empty()</code>
<code>push(e)</code>	<code>push(e)</code>
<code>pop()</code>	<code>pop()</code>
<code>top()</code>	<code>peek()</code>

6.1.2 Una semplice implementazione di pila basata su array

Nella nostra prima implementazione dell'ADT pila, memorizzeremo gli elementi in un array, `data`, con capacità N prefissata, orientando la pila in modo che l'elemento in fondo sia memorizzato nella cella `data[0]`, mentre l'elemento in cima sia `data[t]`, con l'indice t uguale a un'unità in meno della dimensione della pila (come si può vedere nella Figura 6.2).



Figura 6.2: Rappresentazione di una pila memorizzata in un array; l'elemento in cima alla pila si trova nella cella `data[t]`.

Va ricordato che, in Java, gli array iniziano con la cella di indice 0, quindi quando la pila ha i propri elementi memorizzati nelle celle che vanno da `data[0]` a `data[t]`, estremi compresi, ha dimensione $t + 1$. Per convenzione, quando la pila è vuota avrà t uguale a -1 (così, quindi, la sua dimensione, $t + 1$, sarà correttamente uguale a zero). Il Codice 6.2 (nel quale sono stati omessi i commenti secondo lo stile Javadoc, per brevità) riporta un'implementazione completa in Java basata su questa strategia.

Codice 6.2: Un'implementazione dell'interfaccia `Stack` basata su array.

```

1  public class ArrayStack<E> implements Stack<E> {
2      public static final int CAPACITY=1000; // capacità iniziale dell'array
3      private E[] data; // array generico usato per gli elementi
4      private int t = -1; // indice dell'elemento in cima alla pila
5      public ArrayStack() { this(CAPACITY); } // costruisce una pila di capacità standard
6      public ArrayStack(int capacity) { // costruisce una pila di capacità data
7          data = (E[]) new Object[capacity]; // cast sicuro: warning del compilatore
8      }
9      public int size() { return (t + 1); }
10     public boolean isEmpty() { return (t == -1); }

```

```

11  public void push(E e) throws IllegalStateException {
12      if (size() == data.length) throw new IllegalStateException("Stack is full");
13      data[++t] = e;                                // incrementa t prima di usarlo
14  }
15  public E top() {
16      if (isEmpty()) return null;
17      return data[t];
18  }
19  public E pop() {
20      if (isEmpty()) return null;
21      E answer = data[t];
22      data[t] = null;                            // per aiutare il garbage collector
23      t--;
24      return answer;
25  }
26 }
```

Uno svantaggio di questa implementazione di pila basata su array

L'implementazione di una pila mediante array è efficiente e semplice, ma presenta un aspetto negativo: si basa su un array di capacità prefissata, che limita la dimensione effettiva della pila.

Per sua comodità, consentiamo all'utilizzatore della pila di specificarne la capacità come parametro del costruttore (e mettiamo anche a disposizione un costruttore privo di parametri che usa una capacità standard pari a 1000). In quei casi in cui l'utilizzatore della pila è in grado di stimare in modo adeguato il numero di elementi che vi dovranno essere impilati, questa implementazione tramite array è difficile da battere, ma, se la stima è sbagliata, le conseguenze possono essere gravi. Se l'applicazione necessita, in realtà, di uno spazio minore di quello preventivato e assegnato alla pila, c'è uno spreco di memoria; ma, nel caso decisamente peggiore in cui si cerchi di inserire un elemento in una pila che abbia già raggiunto la propria capacità massima, l'implementazione presentata nel Codice 6.2 lancia una `IllegalStateException`, rifiutandosi di memorizzare il nuovo elemento. Quindi, nonostante la sua efficienza e semplicità, l'implementazione di pila basata su array non è necessariamente ideale.

Fortunatamente, vedremo in seguito due approcci che consentono di realizzare una pila senza una tale limitazione nella dimensione e con un'occupazione di spazio sempre proporzionale al numero effettivo di elementi presenti nella pila. Un approccio, descritto nel prossimo sottoparagrafo, memorizza i dati in una lista semplicemente concatenata; nel Paragrafo 7.2.1, invece, proporremo un approccio più avanzato, di nuovo basato su array, che supera i vincoli derivanti dalla capacità prefissata.

Analisi dell'implementazione di pila basata su array

La correttezza dei metodi di questa implementazione basata su array consegue direttamente dalla nostra definizione dell'indice `t`. Si presti attenzione al fatto che, quando si impila un elemento, `t` viene incrementato prima di memorizzare il nuovo elemento nell'array, in modo da usare la prima cella disponibile.

La Tabella 6.2 mostra i tempi d'esecuzione dei metodi di questa implementazione di pila basata su array. Ogni metodo esegue un numero costante di enunciati, che comprendono operazioni aritmetiche, confronti, assegnazioni e invocazioni di `size` e `isEmpty`, due metodi che vengono eseguiti in un tempo costante. Quindi, in questa implementazione dell'ADT pila, ogni metodo viene eseguito in un tempo costante, cioè in un tempo $O(1)$.

Tabella 6.2: Prestazioni di una pila realizzata con un array. Lo spazio utilizzato è $O(N)$, dove N è la dimensione dell'array, decisa nel momento in cui la pila viene creata e indipendente dal numero $n \leq N$ di elementi effettivamente presenti nella pila.

Metodo	Tempo di esecuzione
<code>size</code>	$O(1)$
<code>isEmpty</code>	$O(1)$
<code>top</code>	$O(1)$
<code>push</code>	$O(1)$
<code>pop</code>	$O(1)$

Recupero della memoria non più utilizzata in Java

Vogliamo qui porre l'attenzione su un aspetto interessante che riguarda l'implementazione del metodo `pop` nel Codice 6.2. Abbiamo usato una variabile locale, `answer`, e vi abbiamo memorizzato il riferimento all'elemento che sta per essere eliminato, dopodiché abbiamo volutamente assegnato `null` alla cella `data[t]`, nella riga 22, prima di decrementare `t`. Tecnicamente quest'ultima assegnazione non è necessaria, perché la nostra pila funzionerebbe correttamente anche senza di essa.

Il motivo per cui facciamo in modo che la cella non più utilizzata dalla pila torni ad avere il valore `null` riguarda il meccanismo, in Java, del recupero della memoria non più utilizzata, che si chiama *garbage collection* (letteralmente, "raccolta della spazzatura"), che si occupa di cercare, in memoria, oggetti non più attivi, perché non esiste più alcun riferimento che punti a essi, per poterne così destinare lo spazio a utilizzi futuri (per maggiori dettagli, si veda il Paragrafo 15.1.3). Se continuassimo a conservare nel nostro array un riferimento all'oggetto che è stato eliminato dalla pila, la classe che realizza la pila ignorerebbe questo fatto (eventualmente sovrascrivendo tale riferimento nel momento in cui ulteriori elementi venissero inseriti nella pila), ma, se anche nell'intera applicazione non fosse più attivo alcun riferimento a tale oggetto, il riferimento inutilmente conservato nell'array che realizza la pila impedirebbe al *garbage collector* di Java di recuperare lo spazio di memoria in cui si trova.

Esempio di utilizzo

Concludiamo questo paragrafo illustrando un frammento di codice che crea e utilizza un esemplare della classe `ArrayStack`. In questo esempio, usiamo la classe involucro `Integer` come parametro di tipo effettivo della pila: questa scelta fa in modo che la firma del metodo `push` accetti come parametro un esemplare di `Integer`, mentre il tipo del valore restituito tanto da `top` quanto da `pop` è `Integer`. Ovviamente è possibile usare come parametro di `push` anche un valore del tipo fondamentale `int`, sfruttando le funzionalità di auto-boxing e auto-unboxing di Java (descritte nel Paragrafo 1.3).

Codice 6.3: Esempio di utilizzo della nostra classe `ArrayStack`.

```
Stack<Integer> S = new ArrayStack<>(); // contenuto: ()
S.push(5); // contenuto: (5)
S.push(3); // contenuto: (5,3)
System.out.println(S.size()); // contenuto: (5,3) visualizza 2
System.out.println(S.pop()); // contenuto: (5) visualizza 3
System.out.println(S.isEmpty()); // contenuto: (5) visualizza false
```

```

System.out.println(S.pop());      // contenuto: ()    visualizza 5
System.out.println(S.isEmpty());  // contenuto: ()    visualizza true
System.out.println(S.pop());      // contenuto: ()    visualizza null
S.push(7);                      // contenuto: (7)
S.push(9);                      // contenuto: (7,9)
System.out.println(S.top());      // contenuto: (7,9)  visualizza 9
S.push(4);                      // contenuto: (7,9,4)
System.out.println(S.size());     // contenuto: (7,9,4) visualizza 3
System.out.println(S.pop());      // contenuto: (7,9)  visualizza 4
S.push(6);                      // contenuto: (7,9,6)
S.push(8);                      // contenuto: (7,9,6,8)
System.out.println(S.pop());      // contenuto: (7,9,6) visualizza 8

```

6.1.3 Realizzare una pila con una lista semplicemente concatenata

In questo paragrafo vedremo come si possa facilmente implementare l'interfaccia Stack usando, come spazio di memorizzazione, una lista semplicemente concatenata. Diversamente dalla nostra implementazione basata su array, l'approccio a lista concatenata ha un'occupazione di memoria che è sempre proporzionale al numero di elementi effettivamente presenti nella pila, senza una dimensione massima predefinita.

Per progettare tale implementazione, dobbiamo decidere se la cima della pila debba essere l'estremità iniziale o finale della lista. In questo caso, però, è ovvio che esiste una scelta migliore dell'altra, perché all'estremità iniziale è possibile effettuare inserimenti e rimozioni in un tempo costante: decidendo che la cima della pila sia all'inizio della lista, tutti i metodi verranno eseguiti in un tempo costante.

Lo schema progettuale “adattatore” (adapter)

Lo schema progettuale (*design pattern*) denominato *adattatore* (*adapter*) si applica a qualunque contesto in cui vogliamo modificare una classe esistente in modo che i suoi metodi corrispondano a quelli di una classe o interfaccia correlata, ma diversa. Un metodo generale di applicazione dello schema adattatore consiste nel definire la nuova classe in modo che contenga un esemplare della classe esistente come campo privato, per poi implementare ciascun metodo della nuova classe usando invocazioni di metodi di tale variabile di esemplare privata. Applicando in tal modo lo schema adattatore, si crea una nuova classe che esegue alcune delle funzioni della classe esistente allo stesso modo, ma rendendole pubbliche in modo più conveniente.

Nel contesto dell'ADT pila, possiamo “adattare” la nostra classe SinglyLinkedList, definita nel Paragrafo 3.2.1, per definire la nuova classe LinkedStack, descritta nel Codice 6.4. Questa classe dichiara una variabile privata, list, di tipo SinglyLinkedList, e usa le seguenti corrispondenze tra metodi:

Metodo della pila	Metodo della lista semplicemente concatenata
size()	list.size()
isEmpty()	list.isEmpty()
push(e)	list.addFirst(e)
pop()	list.removeFirst()
top()	list.first()

Codice 6.4: Implementazione di una pila, Stack, usando come spazio di memorizzazione un esemplare di SinglyLinkedList.

```

1  public class LinkedStack<E> implements Stack<E> {
2      private SinglyLinkedList<E> list = new SinglyLinkedList<E>(); // una lista vuota
3      public LinkedStack() { } // la nuova pila si basa su una lista inizialmente vuota
4      public int size() { return list.size(); }
5      public boolean isEmpty() { return list.isEmpty(); }
6      public void push(E element) { list.addFirst(element); }
7      public E top() { return list.getFirst(); }
8      public E pop() { return list.removeFirst(); }
9  }
```

6.1.4 Invertire un array usando una pila

Come conseguenza della strategia LIFO, è possibile usare una pila come strumento generale per invertire una sequenza di dati. Ad esempio, se i valori 1, 2 e 3 vengono impilati in tale ordine, verranno estratti dalla pila nell'ordine 3, 2 e 1.

Illustriamo questo concetto tornando a esaminare il problema dell'inversione della sequenza di elementi memorizzata in un array (per il quale abbiamo già presentato un algoritmo ricorsivo nel Paragrafo 5.3.1). Creiamo una pila vuota come spazio di memorizzazione ausiliario, inseriamo in tale pila tutti gli elementi dell'array e, poi, li estraiamo dalla pila usandoli per sovrascrivere il contenuto delle celle dell'array procedendo dalla prima all'ultima cella. Nel Codice 6.5 presentiamo un'implementazione di questo algoritmo in Java e usiamo tale codice nell'esempio riportato nel Codice 6.6.

Codice 6.5: Un metodo generico che inverte la sequenza degli elementi contenuti in un array di oggetti di tipo E, usando una pila il cui tipo è l'interfaccia Stack<E>.

```

1  /** Un metodo generico che inverte il contenuto di un array. */
2  public static <E> void reverse(E[] a) {
3      Stack<E> buffer = new ArrayStack<E>(a.length);
4      for (int i=0; i < a.length; i++)
5          buffer.push(a[i]);
6      for (int i=0; i < a.length; i++)
7          a[i] = buffer.pop();
8  }
```

Codice 6.6: Un collaudo del metodo reverse usando due array.

```

1  /** Metodo che collauda l'inversione di array. */
2  public static void main(String[] args) {
3      Integer[] a = {4, 8, 15, 16, 23, 42}; // sintassi permessa dall'auto-boxing
4      String[] s = {"Jack", "Kate", "Hurley", "Jin", "Michael"};
5      System.out.println("a = " + Arrays.toString(a));
6      System.out.println("s = " + Arrays.toString(s));
7      System.out.println("Reversing... ");
8      reverse(a);
9      reverse(s);
10     System.out.println("a = " + Arrays.toString(a));
11     System.out.println("s = " + Arrays.toString(s));
12 }
```

L'esecuzione di questo metodo visualizza quanto segue:

```
a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]
```

6.1.5 Corrispondenza tra parentesi e tra marcatori HTML

In questo paragrafo analizzeremo due applicazioni che usano pile, entrambe dedicate alla verifica di coppie di delimitatori corrispondenti. Nella nostra prima applicazione prendiamo in esame espressioni aritmetiche che possono contenere varie coppie di simboli usati per raggruppare sottoespressioni, come:

- Parentesi tonde (*parentheses*): "(" e ")"
- Parentesi graffe (*braces*): "{" e "}"
- Parentesi quadre (*brackets*): "[" e "]"

Ogni simbolo che apre un gruppo (cioè una parentesi aperta, di qualsiasi tipo) deve trovare il proprio corrispondente simbolo di chiusura. Ad esempio, una parentesi quadra aperta, "[", deve trovare una corrispondente parentesi quadra chiusa, "]", come in questa espressione:

$$[(5 + x) - (y + z)]$$

Questi esempi possono essere utili per illustrare ulteriormente il concetto:

- Corretto: ()(){}{((()){})}
- Corretto: (((()){}{((()){}))))
- Sbagliato:)((()){}{((()){})})
- Sbagliato: ({[]}){}
- Sbagliato: (

Lasciamo, però, all'Esercizio R-6.6 il compito di definire con precisione cosa siano i simboli di raggruppamento corrispondenti.

Un algoritmo per trovare corrispondenze tra delimitatori

Un problema importante che va risolto ogni volta che si elaborano espressioni aritmetiche è la verifica della corretta corrispondenza tra i simboli che delimitano le sottoespressioni. Per risolvere questo problema possiamo usare una pila, eseguendo un'unica scansione, da sinistra a destra, della stringa che contiene l'espressione.

Ogni volta che, durante la scansione, incontriamo un simbolo di apertura, lo inseriamo nella pila, mentre ogni volta che incontriamo un simbolo di chiusura, estraiamo un simbolo dalla pila (se non è vuota) e verifichiamo se i due simboli formano una coppia valida. Se arriviamo alla fine della scansione e la pila è vuota, allora l'espressione conteneva parentesi accoppiate correttamente; in caso contrario, nella pila sarà rimasta almeno una parentesi per la quale non è stato trovato il simbolo corrispondente. Se la lunghezza dell'espressione originale è n , l'algoritmo effettuerà al massimo n invocazioni di push e n invocazioni di pop.

Il Codice 6.7 presenta un'implementazione di questo algoritmo in Java. Nello specifico, verifica le coppie di simboli delimitatori (), {} e [], ma potrebbe facilmente essere esteso per gestire ulteriori coppie di simboli. Infatti, nel codice abbiamo definito due stringhe, "({[" e "})]", volutamente coordinate tra loro per rappresentare l'accoppiamento tra i simboli presi in esame dall'algoritmo. Analizzando un carattere della stringa che contiene l'espressione, invochiamo il metodo `indexOf` della classe `String` su queste due stringhe speciali, per determinare se tale carattere sia uno dei delimitatori e, in tal caso, quale. Il metodo `indexOf` restituisce l'indice in corrispondenza del quale un dato carattere viene trovato nella stringa data per la prima volta, partendo dall'inizio (oppure -1 se il carattere non è presente).

Codice 6.7: Metodo che verifica la corrispondenza tra le parentesi di un'espressione aritmetica.

```

1  /** Verifica se nell'espressione data le parentesi sono accoppiate bene. */
2  public static boolean isMatched(String expression) {
3      final String opening = "({[";
4      final String closing = "})]";
5      Stack<Character> buffer = new LinkedStack<>();
6      for (char c : expression.toCharArray()) {
7          if (opening.indexOf(c) != -1)           // è un delimitatore di apertura
8              buffer.push(c);
9          else if (closing.indexOf(c) != -1) { // è un delimitatore di chiusura
10              if (buffer.isEmpty())            // non c'è niente che può corrispondere
11                  return false;
12              if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
13                  return false;             // corrispondenza errata
14          }
15      }
16      return buffer.isEmpty(); // tutte le parentesi aperte hanno trovato corrispondenza?
17  }

```

Corrispondenze in un linguaggio a marcatori (*tag*)

Un'altra applicazione della verifica di corrispondenza tra delimitatori è il controllo di legittimità di un linguaggio a marcatori (*markup language*) come HTML o XML. Il linguaggio HTML è un formato standard per gli ipertesti di Internet, mentre XML è un linguaggio a marcatori estendibile, usato per molti diversi insiemi di dati strutturati. La Figura 6.3 mostra un esempio di documento HTML.

In un documento HTML, le porzioni di testo sono delimitate da appositi *marcatori* (*tag*). Un semplice marcitore di apertura del linguaggio HTML ha la forma “<nome>” e il marcitore di chiusura corrispondente è “</nome>”. Ad esempio, nella Figura 6.3(a) vediamo, nella prima riga, il marcitore `<body>`, mentre il corrispondente marcitore `</body>` chiude il documento. Altri marcatori HTML molto diffusi, che sono stati utilizzati anche nel nostro esempio, sono:

- `<body>`: corpo (*body*) del documento
- `<h1>`: intestazione (*header*) della sezione
- `<center>`: testo centrato (orizzontalmente)
- `<p>`: paragrafo
- ``: elenco numerato (*ordered list*)
- ``: elemento della lista (*list element*)

```

<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>

```

(a)

The Little Boat

The storm tossed the little boat
like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman , who even as
a stowaway now felt that he had
overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

Figura 6.3: Un documento HTML (a) e una sua possibile visualizzazione (b).

In teoria un documento HTML deve avere tutti i marcatori che trovano una corretta corrispondenza, ma la maggior parte dei *browser* (o *navigatori*) sono in grado di tollerare un certo numero di errori. Il Codice 6.8 presenta un metodo Java che verifica le corrispondenze tra marcatori in una stringa che contiene un documento HTML.

Facciamo una scansione da sinistra a destra all'interno dell'intera stringa, usando l'indice *j* per tenere traccia dell'avanzamento. Il metodo *indexOf* della classe *String*, che accetta come secondo parametro opzionale un indice iniziale (da cui partire per fare la sua analisi), individua i caratteri '<' e '>' che definiscono i marcatori. Il metodo *substring*, anch'esso della classe *String*, restituisce la sottostringa che inizia in corrispondenza dell'indice fornito e termina subito prima del secondo indice (che è facoltativo). I marcatori di apertura vengono inseriti sulla pila e messi a confronto con i marcatori di chiusura nel momento in cui vengono estratti dalla pila, esattamente come abbiamo fatto nella verifica di corrispondenza tra parentesi, nel Codice 6.7.

Codice 6.8: Metodo che verifica se un documento HTML contiene marcatori accoppiati correttamente.

```

1  /** Verifica se in una stringa HTML i marcatori aperti e chiusi corrispondono. */
2  public static boolean isHTMLMatched(String html) {
3      Stack<String> buffer = new LinkedStack<>();
4      int j = html.indexOf('<');
5      while (j != -1) {
6          int k = html.indexOf('>', j+1);
7          if (k == -1)
8              return false;
9          String tag = html.substring(j+1, k); // elimina i caratteri < e >
10         if (!tag.startsWith("/"))
11             buffer.push(tag);
12         else { // è un marcatore di chiusura

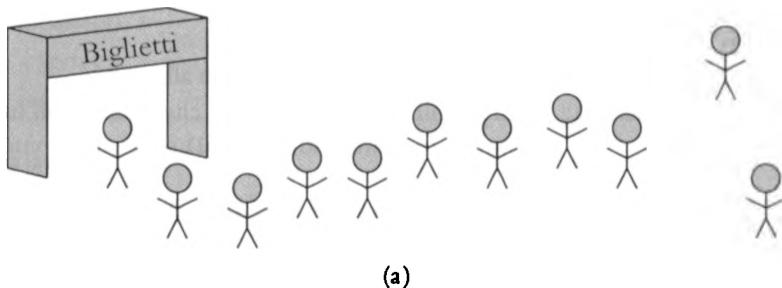
```

```

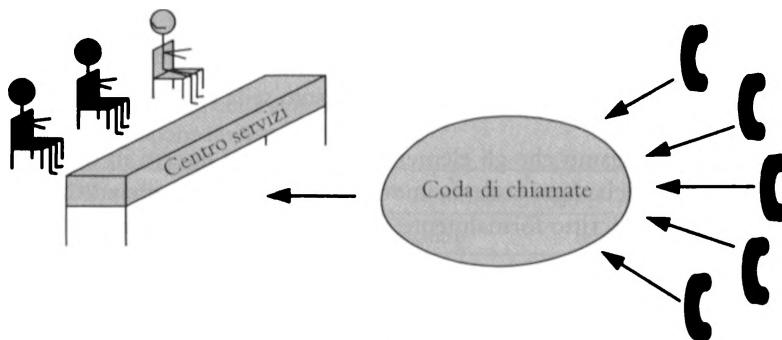
13     if (buffer.isEmpty())
14         return false;           // non ci sono marcatori con cui accoppiarsi
15     if (!tag.substring(1).equals(buffer.pop()))
16         return false;           // marcatore non corrispondente
17   }
18   j = html.indexOf('<', k+1);    // trova il prossimo '<', se ce n'è uno
19 }
20 return buffer.isEmpty(); // tutti i marcatori aperti hanno trovato corrispondenza?
21 }
```

6.2 Code (queue)

Un'altra struttura dati fondamentale è la *coda (queue)*. “Parente stretta” della pila, una coda è un contenitore di oggetti che vengono inseriti e rimossi sulla base di una politica di gestione denominata **FIFO**, cioè *first-in, first-out*: il *primo entrato* sarà il *primo a uscire*. Quindi, è possibile inserire un elemento in qualsiasi momento, ma si può rimuovere soltanto l'elemento che si trova in coda da più tempo.



(a)



(b)

Figura 6.4: Esempio di code FIFO nel mondo reale. (a) Persone in attesa per acquistare biglietti; (b) chiamate telefoniche instradate verso un centro di servizi alla clientela.

Solitamente diciamo che gli elementi entrano nella coda dalla fine e vengono rimossi dall'inizio. La metafora su cui si basa questa terminologia è quella di una coda di persone che attendono di salire su un'attrazione di un parco giochi: le persone che vorrebbero salire

si aggiungono alla fine della coda e quelle che salgono sulla giostra procedono dall'inizio della coda. Esistono molte applicazioni in cui si usano code, alcune delle quali sono illustrate nella Figura 6.4: negozi, teatri, centri di prenotazione e altri servizi simili elaborano le richieste dei clienti seguendo una politica di gestione tipicamente FIFO. Quindi, una coda è la scelta più logica per una struttura dati che gestisca le chiamate a un centro per i servizi alla clientela o la lista d'attesa di un ristorante. Le code FIFO sono utilizzate anche da molti dispositivi di elaborazione, come le stampanti condivise in rete o i server web che devono rispondere alle richieste che arrivano dai navigatori.

6.2.1 Il tipo di dato astratto "coda"

Dal punto di vista formale, il tipo di dato astratto "coda" (*queue*) definisce un contenitore che conserva oggetti in sequenza, nel quale l'accesso agli elementi e la loro rimozione sono limitati al *primo (first)* elemento della coda (cioè nella posizione iniziale, *front*), mentre l'inserimento di nuovi elementi avviene alla *fine (back)* della sequenza. Queste limitazioni garantiscono che venga rispettata la regola che caratterizza la gestione FIFO: nella coda, gli elementi vengono inseriti e rimossi sulla base del principio "il primo entrato sarà il primo a uscire". Il tipo di dato astratto *coda* (*queue*) mette a disposizione i due seguenti metodi di aggiornamento:

enqueue(*e*): Aggiunge l'elemento *e* in fondo alla coda.

dequeue(): Elimina dalla coda l'elemento che si trova all'inizio e lo restituisce (oppure restituisce *null* se la coda è vuota).

Inoltre, il tipo di dato astratto *coda* definisce i seguenti metodi d'accesso (dove il metodo *first* è analogo al metodo *top* della *pila*):

first(): Restituisce il primo elemento della coda, senza eliminarlo (oppure restituisce *null* se la coda è vuota).

size(): Restituisce il numero di elementi presenti nella coda.

isEmpty(): Restituisce *true* se e solo se la coda è vuota.

Per convenzione, ipotizziamo che gli elementi che vengono inseriti in una coda possano essere di qualsiasi tipo e che ogni coda sia vuota nel momento in cui viene creata. Il tipo di dato astratto "coda" è descritto formalmente dall'interfaccia Java riportata nel Codice 6.9.

Codice 6.9: L'interfaccia Queue che definisce l'ADT coda, caratterizzato da un protocollo FIFO per la gestione di inserimenti e rimozioni.

```

1  public interface Queue<E> {
2      /** Restituisce il numero di elementi presenti nella coda. */
3      int size();
4      /** Verifica se la coda è vuota. */
5      boolean isEmpty();
6      /** Inserisce un elemento in fondo alla coda. */
7      void enqueue(E e);
8      /** Restituisce il primo elemento della coda, senza toglierlo (null se è vuota). */
9      E first();

```

```

    /**
     * Elimina e restituisce il primo elemento della coda (null se è vuota).
     * E dequeue();
    }
}

```

Esempio 6.4: La tabella seguente mostra una serie di operazioni eseguite su una coda Q di numeri interi, inizialmente vuota, e gli effetti prodotti su di essa.

Operazione	Valore restituito	first ← Q ← last
enqueue(5)	-	(5)
enqueue(3)	-	(5, 3)
size()	2	(5, 3)
dequeue()	5	(3)
isEmpty()	false	(3)
dequeue()	3	()
isEmpty()	true	()
dequeue()	null	()
enqueue(7)	-	(7)
enqueue(9)	-	(7, 9)
first()	7	(7, 9)
enqueue(4)	-	(7, 9, 4)

L'interfaccia java.util.Queue

La libreria di Java contiene la definizione di un'interfaccia, `java.util.Queue`, la cui funzionalità è analoga a quella che abbiamo visto per il tradizionale ADT coda, ma la documentazione di questa interfaccia, nella libreria, non richiede esplicitamente che le classi che la implementano realizzino soltanto la politica di gestione FIFO. Quando, comunque, una classe che implementa `java.util.Queue` realizza una politica FIFO, allora i metodi dell'interfaccia `java.util.Queue` sono equivalenti a quelli dell'ADT coda elencati nella Tabella 6.3.

Tabella 6.3: Metodi dell'ADT coda e metodi corrispondenti dell'interfaccia `java.util.Queue`, quando la politica di gestione implementata è FIFO.

Il nostro ADT coda	Interfaccia <code>java.util.Queue</code>	
	lancia eccezioni	restituisce un valore speciale
enqueue(e)	add(e)	offer(e)
dequeue()	remove()	poll()
first()	element()	peek()
size()		size()
isEmpty()		isEmpty()

L'interfaccia `java.util.Queue` consente, per la maggior parte delle operazioni, di realizzare due stili diversi, che cambiano rispetto alla modalità di gestione dei casi eccezionali. Quando una coda è vuota, i metodi `remove()` e `element()` lancia un'eccezione di tipo `NoSu-`

`chElementException`, mentre i corrispondenti metodi `poll()` e `peek()` restituiscono `null`. Per implementazioni aventi una capacità limitata, quando la coda è piena il metodo `add` lancia `IllegalStateException`, mentre il metodo `offer` ignora nuovi elementi e restituisce `false`, per segnalare che l'elemento non è stato accettato dalla coda.

6.2.2 Implementazione di coda basata su array

Nel Paragrafo 6.1.2 abbiamo implementato la semantica LIFO dell'ADT pila usando un array (anche se con una capacità fissa), in modo tale che ogni operazione venisse eseguita in un tempo costante. In questo paragrafo vedremo come usare di nuovo un array per realizzare in modo efficiente anche la semantica FIFO, caratteristica dell'ADT coda.

Ipotizziamo che gli elementi inseriti nella coda vengano memorizzati in un array in modo che il primo elemento della coda si trovi in corrispondenza dell'indice 0, il secondo elemento in corrispondenza dell'indice 1, e così via, come rappresentato nella Figura 6.5.

 $N - 1$

Figura 6.5: Utilizzo di un array per memorizzare gli elementi di una coda, in modo che il primo elemento inserito, "A", si trovi nella cella di indice 0, il secondo elemento inserito, "B", nella cella 1, e così via.

In base a questa convenzione, il problema che ci dobbiamo porre, ora, è come si possa implementare l'operazione `dequeue`. L'elemento da rimuovere è memorizzato nella cella di indice 0 dell'array. Una possibile strategia consiste nell'eseguire un ciclo che faccia scorrere tutti gli altri elementi della coda di una cella verso sinistra, in modo che l'elemento iniziale della coda venga di nuovo a trovarsi nella cella 0 dell'array. Sfortunatamente, l'uso di un tale ciclo darebbe luogo a un tempo d'esecuzione $O(n)$ per il metodo `dequeue`.

Possiamo migliorare tale strategia evitando completamente l'utilizzo di un ciclo. Sostituiamo nell'array l'elemento da rimuovere con un riferimento `null` e gestiremo una variabile, f , che rappresenti esplicitamente l'indice dell'elemento che si trova all'inizio della coda, che non sarà più identificato implicitamente dall'indice 0. Tale algoritmo per la realizzazione di `dequeue` verrà eseguito in un tempo $O(1)$. Dopo l'esecuzione di alcune operazioni `dequeue`, questo approccio può portare alla configurazione rappresentata nella Figura 6.6.

 $N - 1$

Figura 6.6: Consentiamo all'inizio della coda di spostarsi verso destra a partire dall'indice 0. In questa configurazione, l'indice f segnala la posizione dell'elemento iniziale della coda.

Tuttavia, anche con questo approccio, così modificato, rimane un problema. Avendo a disposizione un array di capacità N , dovremmo poter memorizzare N elementi prima di raggiungere una condizione "eccezionale", di coda piena, ma, se spostiamo ripetutamente

verso destra la posizione dell'elemento iniziale della coda, la sua posizione finale raggiungerà la fine dell'array anche se gli elementi presenti in coda sono in numero minore di N . Dobbiamo decidere come si possa, con questa configurazione, memorizzare un più elevato numero di elementi.

Utilizzo di un array circolare

Per progettare una versione di coda robusta, consentiamo tanto alla posizione iniziale della coda quanto a quella finale di spostarsi verso destra, "riavvolgendo" il contenuto della coda "attorno" alla fine dell'array, se necessario. Nell'ipotesi che l'array abbia una lunghezza fissa, N , i nuovi elementi vengono accodati in corrispondenza della "fine" della coda, procedendo dall'inizio fino all'indice $N - 1$ e proseguendo, poi, con l'indice 0, l'indice 1, e così via. La Figura 6.7 mostra una coda in cui il primo elemento è F e l'ultimo elemento è R.



Figura 6.7: Modello di coda all'interno di un array circolare: l'ultima cella dell'array contiene un elemento intermedio della coda.

L'implementazione di una tale visione circolare dell'array è relativamente semplice se si usa l'operatore **modulo**, che in Java si indica con il simbolo %. Ricordiamo che il risultato dell'operazione modulo si calcola prendendo il resto della divisione intera tra i suoi operandi. Ad esempio, 14 diviso 3 ha come quoziente 4 e come resto 2, perché $14/3 = 4 + 2/3$. Quindi, in Java, la valutazione dell'espressione $14 \% 3$ fornisce come risultato il resto, 2, mentre $14 / 3$ assume il valore dato dal quoziente, 4.

L'operatore modulo è perfetto per gestire un array circolare. Quando rimuoviamo un elemento dalla coda e vogliamo "far avanzare" l'indice corrispondente alla posizione iniziale della coda, usiamo l'espressione aritmetica $f = (f + 1) \% N$. Per fare un esempio concreto, se abbiamo un array di lunghezza 10 e un indice $f = 7$, possiamo far avanzare la posizione iniziale della coda calcolando $(7+1)\%10$, che vale semplicemente 8, perché 8 diviso 10 ha come quoziente 0 e resto 8. Analogamente, facendo avanzare l'indice 8 otteniamo il valore 9, ma, quando facciamo poi avanzare l'indice 9 (che è l'ultimo indice valido nell'array), calcoliamo $(9+1)\%10$, che ha come valore 0 (perché 10 diviso 10 ha come resto 0).

Un'implementazione di coda in Java

Nel Codice 6.10 presentiamo un'implementazione completa dell'ADT coda usando un array circolare. La classe che realizza la coda gestisce internamente le seguenti variabili di esemplare:

data: un riferimento all'array che memorizza gli elementi.

f: un numero intero che rappresenta l'indice, all'interno dell'array **data**, del primo elemento della coda (nell'ipotesi che non sia vuota).

sz: un numero intero che rappresenta il numero di elementi presenti nella coda (da non confondere con la lunghezza dell'array).

Consentiamo all'utilizzatore di specificare la capacità della coda come parametro facoltativo del costruttore. L'implementazione dei metodi `size` e `isEmpty` è banale, data la presenza del campo `sz`, così come è semplice l'implementazione del metodo `first`. Qualche parola verrà invece spesa per discutere l'implementazione dei metodi `enqueue` e `dequeue`.

Codice 6.10: Implementazione di una coda basata su un array.

```

1  /** Implementazione dell'ADT coda usando un array di lunghezza fissa. */
2  public class ArrayQueue<E> implements Queue<E> {
3      // variabili di esemplare
4      private E[] data;           // array generico che memorizza gli elementi
5      private int f = 0;          // indice dell'elemento iniziale
6      private int sz = 0;         // numero di elementi nella coda
7
8      // costruttori
9      public ArrayQueue() { this(CAPACITY); } // costruisce una coda con capacità standard
10     public ArrayQueue(int capacity) {        // costruisce una coda con capacità data
11         data = (E[]) new Object[capacity];   // cast sicuro: warning del compilatore
12     }
13
14     // metodi
15     /** Restituisce il numero di elementi presenti nella coda. */
16     public int size() { return sz; }
17
18     /** Verifica se la coda è vuota. */
19     public boolean isEmpty() { return (sz == 0); }
20
21     /** Inserisce un elemento in fondo alla coda. */
22     public void enqueue(E e) throws IllegalStateException {
23         if (sz == data.length) throw new IllegalStateException("Queue is full");
24         int avail = (f + sz) % data.length; // usa l'aritmetica modulare
25         data[avail] = e;
26         sz++;
27     }
28
29     /** Restituisce il primo elemento della coda, senza toglierlo (null se è vuota). */
30     public E first() {
31         if (isEmpty()) return null;
32         return data[f];
33     }
34
35     /** Elimina e restituisce il primo elemento della coda (null se è vuota). */
36     public E dequeue() {
37         if (isEmpty()) return null;
38         E answer = data[f];
39         data[f] = null;           // per aiutare il garbage collector
40         f = (f + 1) % data.length;
41         sz--;
42         return answer;
43     }
44     public static final int CAPACITY = 1000;
45 }
```

Aggiungere e rimuovere elementi

Lo scopo del metodo `enqueue` è quello di aggiungere un nuovo elemento alla fine della coda e, per farlo, dobbiamo determinare l'indice corretto in cui posizionare il nuovo elemento. Nonostante non abbiamo introdotto una variabile di esemplare che rappresenti esplicitamente la posizione della fine della coda, possiamo calcolare l'indice della prima posizione libera usando questa formula:

```
avail = (f + sz) % data.length;
```

Si noti che abbiamo usato la dimensione della coda *prima* dell'aggiunta del nuovo elemento. A titolo di verifica, consideriamo una coda con capacità 10, dimensione 3 e primo elemento posizionato nella cella di indice 5, per cui i suoi tre elementi si troveranno nelle celle di indice 5, 6 e 7 e il prossimo elemento aggiunto dovrà essere memorizzato nella cella di indice 8, valore calcolato come $(5+3)\%10$. Per considerare un caso in cui si verifica il ritorno alla prima cella (fenomeno detto *wraparound*), se la coda ha capacità 10, dimensione 3 e il primo elemento si trova nella cella di indice 8, i suoi tre elementi si troveranno nelle celle di indice 8, 9 e 0, e il prossimo elemento aggiunto dovrà essere memorizzato nella cella di indice 1, valore calcolato come $(8+3)\%10$.

Quando viene invocato il metodo `dequeue`, il valore di `f` rappresenta l'indice della cella che contiene il valore che deve essere eliminato e restituito. Per prima cosa memorizziamo in una variabile locale il riferimento all'elemento che dovrà essere restituito, perché poi a tale cella dell'array assegneremo il valore `null`, per aiutare il lavoro del garbage collector. Quindi, aggiorniamo l'indice `f` in modo che tenga conto della rimozione del primo elemento della coda, con la conseguente promozione dell'eventuale secondo elemento che, se presente, diventa il primo della coda. Nella maggior parte dei casi vorremmo semplicemente incrementare di un'unità il valore di tale indice, ma, per via della possibilità di un *wraparound*, ci affidiamo all'aritmetica modulare, calcolando `f = (f + 1)%data.length`, come già detto in precedenza.

Analisi dell'efficienza di una coda basata su array

La Tabella 6.4 mostra i tempi d'esecuzione dei metodi in una realizzazione di coda mediante un array. Come accaduto per la nostra implementazione di pila basata su array, ognuno dei metodi della coda esegue un numero costante di enunciati, che richiedono operazioni aritmetiche, confronti e assegnazioni. Di conseguenza, in questa implementazione ogni metodo viene eseguito in un tempo $O(1)$.

Tabella 6.4: Prestazioni di una coda realizzata con un array. Lo spazio utilizzato è $O(N)$, dove N è la dimensione dell'array, decisa nel momento in cui la coda viene creata e indipendente dal numero $n \leq N$ di elementi effettivamente presenti nella coda.

Metodo	Tempo d'esecuzione
<code>size</code>	$O(1)$
<code>isEmpty</code>	$O(1)$
<code>first</code>	$O(1)$
<code>enqueue</code>	$O(1)$
<code>dequeue</code>	$O(1)$

6.2.3 Coda realizzata con una lista semplicemente concatenata

Come abbiamo fatto per l'ADT pila, possiamo facilmente adattare una lista semplicemente concatenata per implementare l'ADT coda, con prestazioni temporali $O(1)$ per tutte le operazioni e senza alcun vincolo innaturale per la sua capacità. Pare spontaneo orientare la coda in modo che la sua posizione iniziale coincida con l'inizio della lista e, viceversa, la sua posizione finale coincida con la fine della lista, perché l'unica operazione di aggiornamento messa a disposizione dalla lista semplicemente concatenata alla sua estremità finale è l'inserimento. Il Codice 6.11 mostra la nostra implementazione in Java della classe `LinkedQueue`.

Codice 6.11: Implementazione di una coda, `Queue`, usando come spazio di memorizzazione un esemplare di `SinglyLinkedList`.

```

1 public class LinkedQueue<E> implements Queue<E> {
2     private SinglyLinkedList<E> list = new SinglyLinkedList<E>(); // una lista vuota
3     public LinkedQueue() { } // la nuova coda si basa su una lista inizialmente vuota
4     public int size() { return list.size(); }
5     public boolean isEmpty() { return list.isEmpty(); }
6     public void enqueue(E element) { list.addLast(element); }
7     public E first() { return list.first(); }
8     public E dequeue() { return list.removeFirst(); }
9 }
```

Analisi dell'efficienza di una coda basata su lista concatenata

Anche se quando abbiamo presentato, nel Capitolo 3, la nostra implementazione di `SinglyLinkedList` non avevamo ancora parlato di analisi asintotica, dopo un attento esame dei suoi metodi è chiaro che vengono tutti eseguiti in un tempo $O(1)$ nel caso peggiore. Di conseguenza, anche tutti i metodi della classe `LinkedQueue` vengono eseguiti in un tempo $O(1)$ nel caso peggiore.

Abbiamo anche superato il problema di dover specificare una dimensione massima per la coda, come invece avevamo dovuto fare nell'implementazione basata su array, anche se questo vantaggio ha un costo: dato che ogni nodo memorizza un riferimento, `next`, al nodo successivo della lista, in aggiunta al riferimento all'elemento, una lista concatenata occupa uno spazio maggiore, per ciascun elemento, rispetto a un array di riferimenti dimensionato in modo appropriato.

Ancora, nonostante tutti i metodi vengano eseguiti in un tempo costante per entrambe le implementazioni di coda, appare chiaro che le operazioni che coinvolgono le liste concatenate richiedono un numero di operazioni elementari più elevato per ciascuna invocazione. Ad esempio, l'aggiunta di un elemento a una coda realizzata mediante array consiste sostanzialmente del calcolo di un indice usando l'aritmetica modulare, per poi memorizzare l'elemento nella corrispondente cella dell'array e incrementare il contatore che tiene traccia della dimensione della struttura. Con una lista concatenata, un inserimento richiede la creazione e l'inizializzazione di un nuovo nodo, il collegamento di un nodo già esistente a tale nuovo nodo e l'incremento del contatore che, anche in questo caso, tiene traccia della dimensione della struttura. In pratica, tutto questo rende i metodi della realizzazione mediante lista concatenata più onerosi dei corrispondenti metodi dell'implementazione mediante array.

6.2.4 Coda circolare

Nel Paragrafo 3.3 abbiamo realizzato una *lista concatenata circolare* con lo stesso comportamento di una lista semplicemente concatenata e un metodo aggiuntivo, `rotate()`, che sposta il primo elemento alla fine della lista in modo efficiente. Possiamo, ora, generalizzare l'interfaccia `Queue`, definendo l'interfaccia `CircularQueue` avente tale comportamento, come si può vedere nel Codice 6.12.

Codice 6.12: Un'interfaccia Java, `CircularQueue`, che estende l'ADT coda, aggiungendo il metodo `rotate()`.

```

1 public interface CircularQueue<E> extends Queue<E> {
2     /**
3      * Sposta alla fine della coda il suo elemento iniziale.
4      * Se la coda è vuota non fa niente.
5      */
6     void rotate();
7 }
```

Questa interfaccia si può implementare facilmente adattando la classe `CircularlyLinkedList` vista nel Paragrafo 3.3, definendo la nuova classe `LinkedCircularQueue`, che ha un vantaggio rispetto alla normale classe `LinkedList`, in quanto l'invocazione di `Q.rotate()` è implementata in modo più efficiente della combinazione di invocazioni, `Q.enqueue(Q.dequeue())`, dal momento che nessun nodo viene creato, distrutto o ricollegato dall'implementazione dell'operazione di rotazione in una lista concatenata circolare.

La coda circolare è un'eccellente astrazione per quelle applicazioni che gestiscono una disposizione circolare di elementi, come i giochi basati sui turni dei diversi giocatori o la pianificazione a rotazione (*round-robin*) dei processi di elaborazione. Nella parte restante di questo paragrafo vedremo un possibile utilizzo della coda circolare.

Il problema di Josephus

Nel gioco per bambini denominato “hot potato” (*patata bollente*), un gruppo di n bambini si siedono in cerchio e si passano l'un l'altro un oggetto, la “patata”. La patata parte da uno dei bambini presenti nel cerchio e i bambini continuano a passarsela finché il capo (esterno al gioco) suona una campanella: a quel punto il bambino che ha la patata in mano deve abbandonare il gioco e il cerchio, dopo aver affidato la patata al bambino successivo. Dopo questo abbandono, gli altri bambini si spostano un po' e stringono il cerchio. Questa procedura continua finché non rimane un solo bambino, che è il vincitore. Se il conduttore del gioco usa sempre la stessa strategia e suona la campanella dopo k passaggi di mano della patata, con un qualche valore prefissato di k , allora il problema di determinare fin dall'inizio chi sarà il vincitore del gioco dato l'elenco dei partecipanti e le loro posizioni nel cerchio è noto come il *problema di Josephus* (dal nome di un personaggio storico che usò una strategia analoga in un contesto ben più tragico di un gioco per bambini).

Risolvere il problema di Josephus usando una coda

Usando una coda circolare siamo in grado di risolvere il problema di Josephus per un insieme di n elementi, associando la patata all'elemento posto all'inizio della coda e memorizzando gli elementi nella coda seguendo la loro disposizione lungo il cerchio. In questo modo il passaggio della patata da una persona all'altra è equivalente alla rotazione della coda che

porta alla fine il primo elemento. Dopo che questo procedimento è stato eseguito $k - 1$ volte, eliminiamo l'elemento posto all'inizio della coda. Nel Codice 6.13 presentiamo un programma Java completo che risolve il problema di Josephus usando questo approccio, la cui esecuzione richiede un tempo $O(nk)$ (usando tecniche che vanno al di là degli obiettivi di questo libro, è possibile risolvere questo problema in modo più veloce).

Codice 6.13: Un programma Java completo che risolve il problema di Josephus usando una coda circolare.

```

1  public class Josephus {
2      /** Determina il vincitore del problema di Josephus usando una coda circolare. */
3      public static <E> E Josephus(CircularQueue<E> queue, int k) {
4          if (queue.isEmpty()) return null;
5          while (queue.size() > 1) {
6              for (int i=0; i < k-1; i++) // fa ruotare k-1 elementi
7                  queue.rotate();
8              E e = queue.dequeue(); // elimina dalla coda l'elemento iniziale
9              System.out.println(" " + e + " is out");
10         }
11         return queue.dequeue(); // il vincitore
12     }
13
14     /** Costruisce una coda circolare a partire da un array di oggetti. */
15     public static <E> CircularQueue<E> buildQueue(E a[]) {
16         CircularQueue<E> queue = new LinkedCircularQueue<E>();
17         for (int i=0; i<a.length; i++)
18             queue.enqueue(a[i]);
19         return queue;
20     }
21
22     /** Metodo di collaudo */
23     public static void main(String[] args) {
24         String[] a1 = {"Alice", "Bob", "Cindy", "Doug", "Ed", "Fred"};
25         String[] a2 = {"Gene", "Hope", "Irene", "Jack", "Kim", "Lance"};
26         String[] a3 = {"Mike", "Roberto"};
27         System.out.println("First winner is " + Josephus(buildQueue(a1), 3));
28         System.out.println("Second winner is " + Josephus(buildQueue(a2), 10));
29         System.out.println("Third winner is " + Josephus(buildQueue(a3), 7));
30     }
31 }
```

6.3 Coda doppie

Prenderemo ora in esame una struttura dati simile alla coda, che consente però inserimenti e rimozioni a entrambe le proprie estremità (iniziale e finale). Una struttura di questo tipo viene chiamata *coda doppia* o coda a due estremità (*double-ended queue*, spesso abbreviata in *deque*, "parola" che viene solitamente pronunciata in inglese come la parola "deck", per evitare di fare confusione con il metodo *dequeue*, che viene invece pronunciato come la sigla "D.Q.").

Il tipo di dato astratto "coda doppia" è più generale tanto della pila quanto della coda, e questa maggiore generalità può essere utile in alcune applicazioni. Ad esempio, abbiamo

visto come si possa usare una coda per rappresentare la lista d'attesa in un ristorante. A volte, però, può capitare che la prima persona della coda venga fatta entrare, soltanto per scoprire che si era trattato di un errore e che non c'è alcun tavolo disponibile: di solito, ovviamente, in tal caso la persona verrà rimessa *all'inizio* della coda. Può anche capitare che un cliente che si trova alla fine della coda decida di aver aspettato abbastanza e se ne vada, abbandonando la coda (se volessimo tener conto del fatto che i clienti potrebbero decidere di andarsene anche quando si trovano in altre posizioni della coda, avremmo bisogno di una struttura dati ancora più generale).

6.3.1 Il tipo di dato astratto "coda doppia"

Il tipo di dato astratto "coda doppia" (*deque*) è più ricco dei tipi di dati astratti pila e coda. Per poter fornire un'astrazione simmetrica, l'ADT coda doppia è definito con questi metodi di aggiornamento:

- addFirst(e):** Aggiunge l'elemento *e* all'inizio della coda doppia.
- addLast(e):** Aggiunge l'elemento *e* alla fine della coda doppia.
- removeFirst():** Elimina dalla coda doppia l'elemento che si trova all'inizio e lo restituisce (oppure restituisce `null` se la coda doppia è vuota).
- removeLast():** Elimina dalla coda doppia l'elemento che si trova alla fine e lo restituisce (oppure restituisce `null` se la coda doppia è vuota).

Inoltre, il tipo di dato astratto coda doppia definisce i seguenti metodi d'accesso:

- first():** Restituisce il primo elemento della coda doppia, senza eliminarlo (oppure restituisce `null` se la coda doppia è vuota).
- last():** Restituisce l'ultimo elemento della coda doppia, senza eliminarlo (oppure restituisce `null` se la coda doppia è vuota).
- size():** Restituisce il numero di elementi presenti nella coda doppia.
- isEmpty()** Restituisce `true` se e solo se la coda doppia è vuota.

Il tipo di dato astratto "coda doppia" è descritto formalmente dall'interfaccia Java riportata nel Codice 6.14.

Codice 6.14: L'interfaccia Deque che definisce l'ADT coda doppia. Si noti l'uso del parametro di tipo, *E*, che consente alla coda doppia di contenere elementi che siano esemplare di qualunque classe.

```

1  /**
2  * Interfaccia che definisce una coda doppia: un contenitore di elementi con
3  * inserimenti e rimozioni ai due estremi. Semplificata rispetto a java.util.Deque.
4  */
5  public interface Deque<E> {
6      /** Restituisce il numero di elementi presenti nella coda doppia. */
7      int size();
8      /** Verifica se la coda doppia è vuota. */
9      boolean isEmpty();

```

```

10  /** Restituisce il primo elemento, senza toglierlo (null se la coda è vuota). */
11  E first();
12  /** Restituisce l'ultimo elemento, senza toglierlo (null se la coda è vuota). */
13  E last();
14  /** Inserisce un elemento all'inizio della coda doppia. */
15  void addFirst(E e);
16  /** Inserisce un elemento alla fine della coda doppia. */
17  void addLast(E e);
18  /** Elimina e restituisce il primo elemento della coda (null se è vuota). */
19  E removeFirst();
20  /** Elimina e restituisce l'ultimo elemento della coda (null se è vuota). */
21  E removeLast();
22 }

```

Esempio 6.5: La tabella seguente mostra una serie di operazioni eseguite su una coda doppia D di numeri interi, inizialmente vuota, e gli effetti prodotti su di essa.

Operazione	Valore restituito	D
addLast(5)	–	(5)
addFirst(3)	–	(3, 5)
addFirst(7)	–	(7, 3, 5)
first()	7	(7, 3, 5)
removeLast()	5	(7, 3)
size()	2	(7, 3)
removeLast()	3	(7)
removeFirst()	7	()
addFirst(6)	–	(6)
last()	6	(6)
addFirst(8)	–	(8, 6)
isEmpty()	false	(8, 6)
last()	6	(8, 6)

6.3.2 Implementazione di una coda doppia

Il tipo di dato astratto “coda doppia” può essere implementato in modo efficiente memorizzandone gli elementi in un array o in una lista concatenata.

Implementare una coda doppia con un array circolare

Se si decide di usare un array, è preferibile utilizzare una rappresentazione simile a quella vista per la classe `ArrayQueue`, gestendo l’array in modalità circolare e memorizzando l’indice del primo elemento e la dimensione della coda doppia come campi di esemplare (l’indice dell’ultimo elemento verrà calcolato, quando serve, usando l’aritmetica modulare).

Come problema aggiuntivo, occorre evitare l’utilizzo di valori negativi come operandi dell’operatore modulo. Quando si elimina il primo elemento, l’indice corrispondente viene fatto “avanzare” in modalità circolare, usando l’assegnazione $f = (f+1)\%N$, mentre quando

un nuovo elemento viene inserito all'inizio, l'indice corrispondente deve essere, in pratica, decrementato in modalità circolare e usare l'assegnazione $f = (f-1) \% N$ è un errore. Il problema è che, quando f vale 0, l'obiettivo dovrebbe essere quello di "decrementarlo" per portare la posizione iniziale della coda doppia all'altra estremità dell'array, cioè fare assumere all'indice f il valore $N-1$. Tuttavia, il calcolo di quella espressione diventa $-1 \% 10$, che, in Java, produce come risultato il valore -1 . Un modo corretto per decrementare un indice in modalità circolare è, invece, quello di usare l'assegnazione $f = (f-1+N) \% N$. Il termine addizionale N prima del calcolo del resto garantisce che il risultato sarà un numero non negativo. I dettagli relativi a questo approccio saranno affrontati nell'Esercizio P-6.40.

Implementare una coda doppia con una lista doppiamente concatenata

Dato che la coda doppia richiede di effettuare inserimenti e rimozioni tanto all'inizio quanto alla fine, per realizzare tutte le operazioni in modo efficiente con una lista concatenata è più opportuno utilizzare una lista doppiamente concatenata. In effetti, la classe `DoublyLinkedList`, presentata nel Paragrafo 3.4.1, implementa già l'interfaccia `Deque` completa: basta semplicemente aggiungere la dichiarazione "`implements Deque<E>`" alla definizione di quella classe per poterla utilizzare come coda doppia.

Prestazioni delle operazioni di una coda doppia

La Tabella 6.5 mostra i tempi d'esecuzione dei metodi di una coda doppia implementata con un array circolare o con una lista doppiamente concatenata: tutti i metodi sono $O(1)$.

Tabella 6.5: Prestazioni di una coda doppia realizzata con un array circolare o con una lista doppiamente concatenata. Lo spazio utilizzato dall'implementazione mediante array è $O(N)$, dove N è la dimensione dell'array, mentre lo spazio utilizzato dall'implementazione mediante lista doppiamente concatenata è $O(n)$, dove $n \leq N$ è il numero effettivo di elementi presenti nella coda doppia.

Metodo	Tempo d'esecuzione
<code>size, isEmpty</code>	$O(1)$
<code>first, last</code>	$O(1)$
<code>addFirst, addLast</code>	$O(1)$
<code>removeFirst, removeLast</code>	$O(1)$

6.3.3 Code doppie nel Java Collections Framework

Il Java Collections Framework (all'interno della libreria standard di Java) contiene una propria definizione di coda doppia, mediante l'interfaccia `java.util.Deque`, oltre ad alcune sue implementazioni, tra cui una basata sull'uso di un array circolare (`java.util.ArrayDeque`) e una basata sull'utilizzo di una lista doppiamente concatenata (`java.util.LinkedList`). Quindi, se abbiamo bisogno di una coda doppia e non vogliamo implementarla partendo da zero, possiamo semplicemente utilizzare una di queste predefinite.

Come abbiamo visto nel caso di `java.util.Queue` (Paragrafo 6.2.1), l'interfaccia `java.util.Deque` prevede la presenza di metodi duplicati nella funzionalità, che usano tecniche diverse per segnalare casi eccezionali: sono riassunti nella Tabella 6.6.

Tabella 6.6: Metodi dell'ADT coda doppia e metodi corrispondenti dell'interfaccia `java.util.Deque`.

Il nostro ADT coda doppia	Interfaccia <code>java.util.Deque</code>	
	lancia eccezioni	restituisce un valore speciale
<code>first()</code>	<code>getFirst()</code>	<code>peekFirst()</code>
<code>last()</code>	<code>getLast()</code>	<code>peekLast()</code>
<code>addFirst(<i>e</i>)</code>	<code>addFirst(<i>e</i>)</code>	<code>offerFirst(<i>e</i>)</code>
<code>addLast(<i>e</i>)</code>	<code>addLast(<i>e</i>)</code>	<code>offerLast(<i>e</i>)</code>
<code>removeFirst()</code>	<code>removeFirst()</code>	<code>pollFirst()</code>
<code>removeLast()</code>	<code>removeLast()</code>	<code>pollLast()</code>
<code>size()</code>	<code>size()</code>	
<code>isEmpty()</code>	<code>isEmpty()</code>	

Quando si cerca di accedere al primo o all'ultimo elemento di una coda doppia *vuota*, o di eliminarlo, i metodi della colonna centrale della Tabella 6.6 (cioè `getFirst()`, `getLast()`, `removeFirst()` e `removeLast()`) lanciano un'eccezione di tipo `NoSuchElementException`. Invece, i metodi della colonna di destra (cioè `peekFirst()`, `peekLast()`, `pollFirst()` e `pollLast()`) restituiscono semplicemente il riferimento `null`. In modo analogo, quando si cerca di aggiungere un elemento alla fine di una coda che ha raggiunto il proprio limite di capacità, i metodi `addFirst` e `addLast` lanciano un'eccezione, mentre i metodi `offerFirst` e `offerLast` restituiscono `false`.

I metodi che gestiscono le situazioni anomale in modo più silenzioso (cioè senza lanciare eccezioni) sono utili in quelle applicazioni, che rispondono al paradigma *produttore-consumatore*, dove è normale che un componente del programma cerchi un elemento che potrebbe non essere stato ancora posto in coda da un altro componente, oppure provi a inserire un elemento in una coda di dimensione prefissata che potrebbe essere piena. Di converso, i metodi che restituiscono `null` quando la struttura è vuota non sono adeguati per quelle applicazioni in cui `null` potrebbe essere un elemento valido.

6.4 Esercizi

Riepilogo e approfondimento

- R-6.1 Nell'ipotesi che una pila *S* inizialmente vuota abbia eseguito complessivamente 25 operazioni `push`, 12 operazioni `top` e 10 operazioni `pop`, 3 delle quali hanno restituito `null` per segnalare il fatto che la pila era vuota, qual è la dimensione attuale di *S*?
- R-6.2 Se la pila dell'esercizio precedente fosse stata un esemplare della classe `ArrayList`, definita nel Codice 6.2, quale sarebbe stato il valore finale della variabile di esemplare *t*?
- R-6.3 Quali valori vengono restituiti durante la seguente sequenza di operazioni su una pila, eseguita a partire da una pila vuota? `push(5)`, `push(3)`, `pop()`, `push(2)`, `push(8)`, `pop()`, `pop()`, `push(9)`, `push(1)`, `pop()`, `push(7)`, `push(6)`, `pop()`, `push(4)`, `pop()`, `pop()`.

- R-6.4 Implementare un metodo, avente la firma `transfer(S, T)`, che trasferisca tutti gli elementi della pila `S` nella pila `T`, in modo che l'elemento che inizialmente si trova in cima a `S` sia il primo a essere inserito in `T`, e che l'elemento che inizialmente si trova in fondo a `S` sia, alla fine, in cima a `T`.
- R-6.5 Progettare un metodo ricorsivo per eliminare tutti gli elementi da una pila.
- R-6.6 Dare una definizione precisa e completa del concetto di corrispondenza tra simboli di raggruppamento di sottoespressioni in un'espressione aritmetica. La definizione può essere ricorsiva.
- R-6.7 Nell'ipotesi che una coda `Q` inizialmente vuota abbia eseguito complessivamente 32 operazioni `enqueue`, 10 operazioni `first` e 15 operazioni `dequeue`, 5 delle quali hanno restituito `null` per segnalare il fatto che la coda era vuota, qual è la dimensione attuale di `Q`?
- R-6.8 Se la coda dell'esercizio precedente fosse stata un esemplare della classe `ArrayList`, definita nel Codice 6.10, con capacità uguale a 30 e mai superata, quale sarebbe stato il valore finale della variabile di esemplare `f`?
- R-6.9 Quali valori vengono restituiti durante la seguente sequenza di operazioni su una coda, eseguita a partire da una coda vuota? `enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue()`.
- R-6.10 Progettare un semplice adattatore che implementi l'ADT pila usando un esemplare di coda doppia come spazio di memorizzazione.
- R-6.11 Progettare un semplice adattatore che implementi l'ADT coda usando un esemplare di coda doppia come spazio di memorizzazione.
- R-6.12 Quali valori vengono restituiti durante la seguente sequenza di operazioni su una coda doppia, eseguita a partire da una coda doppia vuota? `addFirst(3), addLast(8), addLast(9), addFirst(1), last(), isEmpty(), addFirst(2), removeLast(), addLast(7), first(), last(), addLast(4), size(), removeFirst(), removeFirst()`.
- R-6.13 Ipotizzare di disporre di una coda doppia `D` contenente i numeri (1, 2, 3, 4, 5, 6, 7, 8), in questo ordine, e di una coda `Q` inizialmente vuota. Progettare un frammento di codice che usi soltanto `D` e `Q` (e nessun'altra variabile) e produca come risultato la sequenza (1, 2, 3, 5, 4, 6, 7, 8) memorizzata, in questo ordine, in `D`.
- R-6.14 Risolvere nuovamente l'esercizio precedente usando la stessa coda doppia `D` e una pila `S`, inizialmente vuota.
- R-6.15 Aggiungere all'implementazione di `ArrayList` un nuovo metodo, `rotate()`, che abbia lo stesso comportamento della combinazione di invocazioni `enqueue(dequeue())`, ma sia più efficiente delle due invocazioni separate (ad esempio, perché non c'è bisogno di modificare la dimensione della coda).

Creatività

- C-6.16 Ipotizzare che Alice abbia scelto tre numeri interi distinti e li abbia inseriti in ordine casuale in una pila `S`. Scrivere un breve frammento di codice, semplice (senza cicli né ricorsione), che usi un solo confronto e una sola variabile, `x`, memorizzando in essa, con probabilità 2/3, il maggiore dei tre numeri scelti da Alice. Dimostrare la correttezza del metodo proposto.

- C-6.17 Spiegare come si possa usare il metodo `transfer`, descritto nell'Esercizio R-6.4, e due pile temporanee per sostituire il contenuto di una pila `S` con i suoi stessi elementi, in ordine inverso.
- C-6.18 Nel Codice 6.8 abbiamo ipotizzato che i marcatori di apertura, in HTML, abbiano il formato `<nome>`, come ``. Più in generale, il linguaggio HTML consente di inserire attributi facoltativi all'interno del marcatore di apertura, secondo il formato generale `<nome attributo1="valore1" attributo2="valore2">`; ad esempio, a una tabella (`table`) si può aggiungere un bordo (`border`) e una spaziatura di riempimento nelle singole celle (`cell padding`), usando il marcatore `<table border="3" cellpadding="5">`. Modificare il Codice 6.8 in modo che possa gestire in modo appropriato i marcatori, anche se un marcatore di apertura contiene attributi.
- C-6.19 La *notazione postfissa* è un sistema non ambiguo per scrivere espressioni aritmetiche senza usare parentesi. È definita in modo che se " $(exp_1) \text{ op } (exp_2)$ " è una normale espressione contenente parentesi, il cui operatore è `op`, la sua versione postfissa è " $pexp_1 \text{ } pexp_2 \text{ } op$ ", dove `pexp1` è la versione postfissa dell'espressione `exp1` e `pexp2` è la versione postfissa dell'espressione `exp2`. La versione postfissa di un numero o di una singola variabile è proprio quel numero o quella variabile. Così, ad esempio, la versione postfissa di " $((5 + 2) * (8 - 3)) / 4$ " è " $5 \text{ } 2 + 8 \text{ } 3 - * \text{ } 4 /$ ". Descrivere un modo non ricorsivo per valutare un'espressione in notazione postfissa.
- C-6.20 Date tre pile non vuote, `R`, `S` e `T`, descrivere una sequenza di operazioni che, alla fine, memorizzino in `S`, al di sotto di tutti gli elementi originariamente presenti in `S`, tutti gli elementi originariamente memorizzati in `T`, con gli elementi di entrambi tali insiemi che abbiano preservato il loro ordinamento originario. Sempre al termine della sequenza di operazioni, `R` deve avere lo stesso contenuto che aveva all'inizio, nello stesso ordine. Se, ad esempio, dal basso in alto, il contenuto originario delle pile è `R = (1, 2, 3)`, `S = (4, 5)` e `T = (6, 7, 8, 9)`, al termine la situazione dovrà essere `R = (1, 2, 3)` e `S = (6, 7, 8, 9, 4, 5)`.
- C-6.21 Descrivere un algoritmo non ricorsivo per elencare tutte le permutazioni dei numeri $\{1, 2, \dots, n\}$ usando esplicitamente una pila.
- C-6.22 Alice dispone di tre pile basate su array, `A`, `B` e `C`, con `A` che ha capacità uguale a 100, `B` ha capacità 5 e `C` ha capacità 3. Inizialmente `A` è piena e `B` e `C` sono vuote. Sfortunatamente, chi ha progettato il codice della classe di cui le tre pile sono esemplari ha reso privati i metodi `push` e `pop`. L'unico metodo che Alice può utilizzare è un metodo statico, `dump(S, T)`, che (invocando ripetutamente i metodi privati `pop` e `push`) trasferisce elementi dalla pila `S` alla pila `T` finché non succede che `S` diventa vuota oppure `T` diventa piena. Così, ad esempio, partendo dalla configurazione iniziale citata e invocando `dump(A, C)`, si ottiene come risultato una situazione in cui `A` contiene 97 elementi e `C` ne contiene 3. Descrivere una sequenza di operazioni `dump` che, partendo dalla configurazione iniziale citata, giunga a una situazione finale in cui `B` contiene esattamente 4 elementi.
- C-6.23 Spiegare come si possano usare una pila `S` e una coda `Q` per generare, senza ricorsione, tutti i possibili sottoinsiemi di un insieme `T` avente n elementi.
- C-6.24 Immaginando di disporre di una pila `S` contenente n elementi e di una coda `Q` inizialmente vuota, descrivere come si possa usare `Q` per scandire gli elementi di `S` al fine di verificare se contiene un determinato elemento `x`, con il vincolo

aggiuntivo di dover riportare gli elementi in S alla loro configurazione originaria. Si può usare soltanto S , Q e un numero costante di altre variabili di tipi primitivi.

- C-6.25 Descrivere come implementare l'ADT pila usando un'unica coda come variabile di esemplare e, all'interno dei corpi dei metodi, soltanto una quantità costante di memoria locale. Qual è il tempo d'esecuzione dei metodi `push`, `pop` e `top`?
- C-6.26 Realizzando la classe `ArrayList`, nella riga 5 del Codice 6.10 abbiamo inizializzato a a 0. Cosa sarebbe successo se avessimo inizializzato quel campo a un diverso valore positivo? E se l'avessimo inizializzato al valore -1?
- C-6.27 Implementare il metodo `clone()` nella classe `ArrayList` (rivedendo il Paragrafo 3.6 che tratta della clonazione di strutture dati).
- C-6.28 Implementare il metodo `clone()` nella classe `ArrayList` (rivedendo il Paragrafo 3.6 che tratta della clonazione di strutture dati).
- C-6.29 Implementare nella classe `LinkedList<E>` un metodo avente la firma `concatenate(LinkedList<E> Q2), che prenda tutti gli elementi di Q_2 e li aggiunga alla fine della coda che costituisce il suo parametro implicito. L'operazione deve essere eseguita in un tempo $O(1)$ e deve lasciare vuota la coda Q_2 .`
- C-6.30 Descrivere mediante pseudocodice un'implementazione basata su array dell'ADT coda doppia. Qual è il tempo d'esecuzione di ciascuna operazione?
- C-6.31 Descrivere come si possa implementare l'ADT coda doppia usando due pile come uniche variabili di esemplare. Qual è il tempo d'esecuzione dei metodi?
- C-6.32 Date due pile S e T non vuote e una coda doppia D , descrivere come si possa usare D per fare in modo che S memorizzi tutti gli elementi di T al di sotto degli elementi originariamente contenuti in S , con entrambi gli insiemi di elementi che preservano il proprio ordinamento originario.
- C-6.33 Alice dispone di due code circolari, C e D , che possono memorizzare numeri interi. Bob fornisce ad Alice 50 numeri dispari e 50 numeri pari, in modo che tutti i 100 numeri vengano memorizzati in C e in D . Poi, fanno un gioco durante la quale Bob sceglie C o D a caso e applica per un numero casuale di volte il metodo `rotate()` alla coda scelta. Se l'ultimo numero che viene fatto circolare al termine del gioco è dispari, Bob vince, altrimenti vince Alice. In che modo Alice può posizionare i numeri interi nelle code, all'inizio del gioco, in modo da rendere massima la sua probabilità di vittoria? Qual è tale probabilità?
- C-6.34 Bob possiede quattro mucche e le vuol far passare sopra un ponte, ma ha un solo giogo, al quale può legare al massimo due sole mucche, una di fianco all'altra. Il giogo è troppo pesante perché Bob lo possa trasportare al di là del ponte, ma vi può legare (e slegare) mucche istantaneamente, senza perdite di tempo. Delle sue quattro mucche, Mazie può attraversare il ponte in 2 minuti, Daisy può farlo in 4 minuti, Crazy in 10 minuti e Lazy in 20. Quando due mucche sono legate al giogo, naturalmente devono viaggiare alla velocità di quella più lenta. Spiegare come Bob possa trasportare tutte le sue mucche al di là del ponte in 34 minuti.

Progettazione

- P-6.35 Realizzare un programma che sia in grado di acquisire un'espressione in notazione postfissa (si veda l'Esercizio C-6.19) e di visualizzarne il valore.

- P-6.36 Quando viene venduta un'azione ordinaria di una società, il *guadagno in conto capitale* (*capital gain*, che a volte può essere in effetti una perdita, *capital loss*) è la differenza tra il prezzo di vendita e il prezzo pagato originariamente per l'acquisto dell'azione. Questa regola è di facile comprensione quando viene applicata a una singola azione, ma quando si vendono molte azioni acquistate durante un lungo periodo di tempo, bisogna identificare quali siano le azioni che vengono effettivamente vendute. Un principio standard di contabilità che permette, in casi come questo, di identificare quali siano le azioni vendute prevede di utilizzare un protocollo FIFO: le azioni vendute sono quelle che si possiedono da più tempo (e, in effetti, questo è proprio l'algoritmo utilizzato da molti programmi di gestione della contabilità personale). Supponiamo, ad esempio, di aver comprato 100 azioni a \$20 l'una nel giorno 1, 20 azioni a \$24 nel giorno 2, 200 azioni a \$36 nel giorno 3; quindi, nel giorno 4 vendiamo 150 azioni a \$30 l'una. Applicando il protocollo FIFO, delle 150 azioni vendute, 100 erano state comprate nel giorno 1, 20 nel giorno 2 e 30 nel giorno 3. Il guadagno, quindi, sarebbe $100 \cdot 10 + 20 \cdot 6 + 30 \cdot (-6)$, cioè \$940. Scrivere un programma che acquisisca in ingresso una sequenza di transazioni, nella forma "buy x share(s) at \$ y each" (per indicare l'acquisto di x azioni a \$ y l'una) oppure "sell x share(s) at \$ y each" (per indicare la vendita di x azioni a \$ y l'una), nell'ipotesi che le transazioni avvengano in giorni consecutivi e che i valori x e y siano numeri interi. Data una tale sequenza in ingresso, il programma deve calcolare il guadagno (o la perdita) totale derivante dall'intera sequenza, usando il protocollo FIFO.
- P-6.37 Progettare un tipo di dato astratto per una doppia pila a due colori, costituita da due pile, una "rossa" (*red*) e una "blu" (*blue*), con versioni "colorate" di tutte le normali operazioni dell'ADT pila. Ad esempio, questo ADT deve mettere a disposizione le operazioni *redPush* e *bluePush*. Descrivere un'implementazione efficiente di questo ADT usando un unico array, la cui capacità sia impostata a un valore N che si ipotizza essere sempre maggiore della somma delle dimensioni delle due pile, rossa e blu.
- P-6.38 Nella parte iniziale del Paragrafo 6.1 abbiamo ricordato che spesso le pile vengono usate per fornire il supporto per l'annullamento di operazioni (*undo*) eseguite in applicazioni come i browser web o gli editor di testo. Nonostante tale supporto possa essere fornito mediante una pila priva di limitazioni sulla dimensione, molte applicazioni pongono un *limite* alla quantità di operazioni che possono essere annullate, usando una pila di dimensioni prefissate. Quando viene invocato il metodo *push* e la pila è piena, invece di lanciare un'eccezione, la semantica maggiormente implementata in queste applicazioni consiste nell'accettare l'elemento inserito, ponendolo in cima alla pila e facendone uscire l'elemento meno recente, che ovviamente si trova in fondo alla pila, per fare spazio. Si parla, in questo caso, di *leakage*, cioè di "perdita" di un elemento in fondo alla pila. Usando un array circolare, implementare una classe, *LeakyStack*, che rispetti questa astrazione.
- P-6.39 Risolvere nuovamente il problema precedente usando, per la memorizzazione degli elementi, una lista semplicemente concatenata, specificando la capacità massima della pila come parametro fornito al costruttore.
- P-6.40 Progettare un'implementazione completa dell'ADT coda doppia usando un array di capacità prefissata, in modo tale che tutti i metodi di aggiornamento vengano eseguiti in un tempo $O(1)$.

Note

I testi di Aho, Hopcroft e Ullman [5, 6], ormai classici, sono stati i primi a seguire l'ap-
proccio di una prima definizione delle strutture dati in termini di tipi di dati astratti, per
passare soltanto dopo alle implementazioni concrete. Gli Esercizi C-6.22, C-6.33 e C-6.34
sono simili a domande che vengono poste durante i colloqui di assunzione in una famosa
azienda che produce software. Per approfondire lo studio dei tipi di dati astratti, consigliamo
di consultare Liskov e Guttag [67] e Demurjian [28].

7

Liste e iteratori

7.1 Liste

Nel Capitolo 6 abbiamo introdotto i tipi di dati astratti pila, coda e coda doppia, discutendo come possano essere implementati concretamente usando come supporto di memorizzazione un array o una lista concatenata. Ciascuno di quei tipi di dati astratti rappresenta una sequenza di elementi ordinata linearmente: la coda doppia è il tipo più generale dei tre, anche se, comunque, consente solamente operazioni di inserimento e rimozione all'inizio o alla fine della sequenza.

In questo capitolo vedremo alcuni altri tipi di dati astratti che rappresentano sequenze lineari di elementi, ma con un supporto più generale all'inserimento o alla rimozione in qualsiasi posizione nella sequenza. Tuttavia, è piuttosto difficile progettare un'unica astrazione di comportamento che si adatti bene a un'efficiente implementazione tanto con un array quanto con una lista concatenata, visto che queste due strutture dati fondamentali hanno una natura molto diversa.

All'interno di un array, le posizioni sono ben identificate da un numero intero, detto *indice*. Ricordiamo che l'indice di un elemento *e* in una sequenza è uguale al numero di elementi che precedono *e* in quella stessa sequenza. In base a questa definizione, il primo elemento di una sequenza ha indice 0 e l'ultimo ha indice $n - 1$, se n è il numero totale di elementi presenti nella sequenza. Il concetto di indice di un elemento è ben definito anche in una lista concatenata, anche se, come vedremo, il suo utilizzo non è altrettanto comodo, perché non c'è modo di accedere in modo efficiente a un elemento tramite il suo indice senza attraversare una porzione della lista concatenata la cui dimensione dipende dal valore dell'indice.

Ciò detto, nella libreria di Java troviamo la definizione di un'interfaccia piuttosto generale, `java.util.List`, che comprende i seguenti metodi basati su indice (e anche altri):

- size():** Restituisce il numero di elementi presenti nella lista.
- isEmpty():** Restituisce `true` se e solo se la lista è vuota.
- get(*i*):** Restituisce l'elemento avente indice *i* nella lista; si verifica una condizione d'errore se *i* non appartiene all'intervallo $[0, \text{size()} - 1]$.
- set(*i*, *e*):** Nella lista, sostituisce con *e* l'elemento avente indice *i* e restituisce l'elemento che è stato sostituito; si verifica una condizione d'errore se *i* non appartiene all'intervallo $[0, \text{size()} - 1]$.
- add(*i*, *e*):** Inserisce *e* come nuovo elemento nella lista in modo che abbia indice *i*, spostando di una posizione verso indici maggiori tutti gli elementi successivi; si verifica una condizione d'errore se *i* non appartiene all'intervallo $[0, \text{size()}]$.
- remove(*i*):** Elimina e restituisce l'elemento della lista avente indice *i*, spostando di una posizione verso indici minori tutti gli elementi successivi; si verifica una condizione d'errore se *i* non appartiene all'intervallo $[0, \text{size()} - 1]$.

Osserviamo che l'indice associato a un elemento della lista può cambiare nel tempo, per effetto di altri elementi che vengano inseriti o rimossi in posizioni precedenti. Vogliamo anche segnalare che l'intervallo di indici validi per il metodo `add` contiene la dimensione attuale della lista, nel qual caso il nuovo elemento diventa ovviamente l'ultimo.

L'Esempio 7.1 mostra una serie di operazioni effettuate su un esemplare di lista, mentre il Codice 7.1 presenta una definizione formale della nostra versione semplificata dell'interfaccia `List`, dove usiamo un'eccezione di tipo `IndexOutOfBoundsException` per segnalare il passaggio di un indice non valido come parametro.

Esempio 7.1: La tabella seguente mostra una serie di operazioni eseguite su una lista di caratteri, inizialmente vuota, e gli effetti prodotti su di essa.

Operazione	Valore restituito	Contenuto della lista
<code>add(0, A)</code>	-	(A)
<code>add(0, B)</code>	-	(B, A)
<code>get(1)</code>	A	(B, A)
<code>set(2, C)</code>	"error"	(B, A)
<code>add(2, C)</code>	-	(B, A, C)
<code>add(4, D)</code>	"error"	(B, A, C)
<code>remove(1)</code>	A	(B, C)
<code>add(1, D)</code>	-	(B, D, C)
<code>add(1, E)</code>	-	(B, E, D, C)
<code>get(4)</code>	"error"	(B, E, D, C)
<code>add(4, F)</code>	-	(B, E, D, C, F)
<code>set(2, G)</code>	D	(B, E, G, C, F)
<code>get(2)</code>	G	(B, E, G, C, F)

Codice 7.1: Una versione semplificata dell'interfaccia `java.util.List`.

```

1  /** Una versione semplificata dell'interfaccia java.util.List. */
2  public interface List<E> {
3      /** Restituisce il numero di elementi presenti nella lista. */
4      int size();
5
6      /** Restituisce true se e solo se la lista è vuota. */
7      boolean isEmpty();
8
9      /** Restituisce l'elemento corrispondente all'indice i, senza eliminarlo. */
10     E get(int i) throws IndexOutOfBoundsException;
11
12     /** Sostituisce con e l'elemento di indice i; restituisce l'elemento sostituito. */
13     E set(int i, E e) throws IndexOutOfBoundsException;
14
15     /** Inserisce e come elemento di indice i, spostando gli elementi successivi. */
16     void add(int i, E e) throws IndexOutOfBoundsException;
17
18     /** Elimina e restituisce l'elemento di indice i, spostando i successivi. */
19     E remove(int i) throws IndexOutOfBoundsException;
20 }

```

7.2 Liste con indice

Una scelta ovvia per l'implementazione del tipo di dato astratto “lista” è l'utilizzo di un array, A , in modo che $A[i]$ memorizzi (un riferimento a) l'elemento avente indice i . Come ipotesi di lavoro iniziale, useremo un array di capacità prefissata, ma nel Paragrafo 7.2.1 descriveremo una tecnica più avanzata che, a tutti gli effetti, consente di disporre di una lista basata su array con capacità non limitata: quel tipo di lista, non limitata nelle dimensioni, viene chiamata, in Java, *lista con indice* (*array list*) o *vettore* (*vector*, soprattutto in C++ e nelle prime versioni di Java).

Usando una rappresentazione basata sull'array A , i metodi `get(i)` e `set(i)` sono di facile implementazione, potendo accedere direttamente alla cella $A[i]$ (nell'ipotesi che i sia un indice valido). I metodi `add(i, e)` e `remove(i)` richiedono un tempo d'esecuzione maggiore, dal momento che è necessario far scorrere elementi verso indici maggiori o, rispettivamente, minori per preservare la regola che prevede di avere sempre un elemento memorizzato nella cella i dell'array se i è un indice valido nella lista (si veda la Figura 7.1). Nel Codice 7.2 e 7.3 è presentata la nostra implementazione iniziale della classe `ArrayList`.

Codice 7.2: Semplice implementazione della classe `ArrayList`, con capacità limitata (prosegue nel Codice 7.3).

```

1  public class ArrayList<E> implements List<E> {
2      // variabili di esemplare
3      public static final int CAPACITY=16; // capacità predefinita dell'array
4      private E[] data;                  // array generico per memorizzare gli elementi
5      private int size=0;                // numero di elementi nella lista
6      // costruttori
7      public ArrayList() { this(CAPACITY); } // costruisce una lista con capacità standard
8      public ArrayList(int capacity) {       // costruisce una lista con capacità data
9          data = (E[]) new Object[capacity];
10 }

```

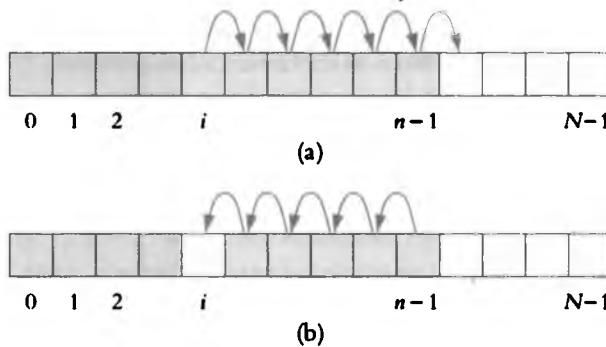


Figura 7.1: Implementazione basata su array di una lista con indice che contiene n elementi: (a) scorrimento "verso l'alto" (cioè verso indici maggiori) conseguente a un inserimento in corrispondenza dell'indice i ; (b) scorrimento "verso il basso" dopo una rimozione avvenuta in corrispondenza dell'indice i .

Codice 7.3: Semplice implementazione della classe `ArrayList`, con capacità limitata (continua dal Codice 7.2).

```

11 // metodi pubblici
12 /** Restituisce il numero di elementi presenti nella lista. */
13 public int size() { return size; }
14 /** Restituisce true se e solo se la lista è vuota. */
15 public boolean isEmpty() { return size == 0; }
16 /** Restituisce l'elemento corrispondente all'indice  $i$ , senza eliminarlo. */
17 public E get(int i) throws IndexOutOfBoundsException {
18     checkIndex(i, size);
19     return data[i];
20 }
21 /** Sostituisce con  $e$  l'elemento di indice  $i$ ; restituisce l'elemento sostituito. */
22 public E set(int i, E e) throws IndexOutOfBoundsException {
23     checkIndex(i, size);
24     E temp = data[i];
25     data[i] = e;
26     return temp;
27 }
28 /** Inserisce  $e$  come elemento di indice  $i$ , spostando gli elementi successivi. */
29 public void add(int i, E e) throws IndexOutOfBoundsException,
30                             IllegalStateException {
31     checkIndex(i, size + 1);
32     if (size == data.length)          // non c'è spazio sufficiente
33         throw new IllegalStateException("Array is full");
34     for (int k=size-1; k >= i; k--) // fa scorrere verso destra a partire da destra
35         data[k+1] = data[k];
36     data[i] = e;                    // ha fatto spazio per il nuovo elemento
37     size++;
38 }
39 /** Elimina  $e$  e restituisce l'elemento di indice  $i$ , spostando i successivi. */
40 public E remove(int i) throws IndexOutOfBoundsException {
41     checkIndex(i, size);
42     E temp = data[i];
43     for (int k=i; k < size-1; k++) // fa scorrere per "chiudere il buco"
44         data[k] = data[k+1];
45     data[size-1] = null;           // per aiutare il garbage collector

```

```

46     size--;
47     return temp;
48   }
49   // metodo ausiliario
50   /** Controlla se l'indice dato appartiene all'intervallo [0, n-1]. */
51   protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52     if (i < 0 || i >= n)
53       throw new IndexOutOfBoundsException("Illegal index: " + i);
54   }
55 }
```

Prestazioni di una semplice implementazione basata su array

La Tabella 7.1 mostra i tempi d'esecuzione nel caso peggiore dei metodi di una lista con indice avente n elementi e realizzata mediante un array. I metodi `isEmpty`, `size`, `get` e `set` sono ovviamente $O(1)$, ma i metodi di inserimento e rimozione possono richiedere molto più tempo. Nello specifico, `add(i, e)` richiede un tempo $O(n)$. Il caso peggiore per questa operazione si verifica quando *i* vale 0, perché tutti gli elementi presenti nella lista, che sono n , devono essere spostati in avanti di una posizione. Un ragionamento analogo si applica al metodo `remove(i)`, che viene eseguito in un tempo $O(n)$, perché nel caso peggiore, quando *i* vale 0, bisogna spostare all'indietro tutti gli $n - 1$ elementi rimasti nella lista dopo la rimozione. Ipotizzando che ogni possibile indice abbia la stessa probabilità di essere passato come argomento a una di queste operazioni, il loro tempo d'esecuzione medio è $O(n)$, perché in media bisogna spostare $n/2$ elementi.

Tabella 7.1: Prestazioni di una lista con indice avente n elementi realizzata mediante un array di capacità prefissata.

Metodo	Tempo d'esecuzione
<code>size()</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>get(<i>i</i>)</code>	$O(1)$
<code>set(<i>i</i>, <i>e</i>)</code>	$O(1)$
<code>add(<i>i</i>, <i>e</i>)</code>	$O(n)$
<code>remove(<i>i</i>)</code>	$O(n)$

Esaminando con maggiore attenzione i metodi `add(i, e)` e `remove(i)`, osserviamo che entrambi vengono eseguiti in un tempo $O(n - i + 1)$, perché soltanto gli elementi di indice non minore di *i* devono essere spostati. Quindi, l'inserimento e la rimozione di un elemento alla fine di una lista con indice, usando rispettivamente i metodi `add(n, e)` e `remove(n - 1)`, richiedono un tempo $O(1)$. Questa osservazione ha, poi, un'interessante conseguenza nel caso in cui si voglia adattare questo ADT “lista con indice” all’ADT “coda doppia” visto nel Paragrafo 6.3.1. Se si fa la cosa “ovvia” e si memorizzano gli elementi di una coda doppia in modo che il primo elemento corrisponda all’indice 0 e l’ultimo elemento corrisponda all’indice $n - 1$, allora i metodi `addLast` e `removeLast` della coda doppia vengono eseguiti in un tempo $O(1)$, mentre i metodi `addFirst` e `removeFirst` richiedono un tempo $O(n)$.

In realtà, con poca fatica si può realizzare un’implementazione dell’ADT “lista con indice” basata su array che abbia prestazioni temporali $O(1)$ anche per inserimenti e rimozioni che avvengano in corrispondenza dell’indice 0, così come gli inserimenti e le rimozioni

alla fine della lista. Per ottenere questo risultato dobbiamo, però, rinunciare alla regola che avevamo imposto, che prevedeva di avere l'elemento di indice i della lista memorizzato nella cella di indice i dell'array, perché dobbiamo usare un approccio ad array circolare come quello che abbiamo visto nel Paragrafo 6.2 per implementare una coda. Lasciamo i dettagli di questa implementazione all'Esercizio C-7.25.

7.2.1 Array dinamici

L'implementazione di `ArrayList` che abbiamo visto nel Codice 7.2 e 7.3 (così come quelle di pila, coda e coda doppia viste nel Capitolo 6) ha un forte limite: la capacità massima del contenitore va dichiarata in fase di costruzione e rimane fissa, provocando il lancio di un'eccezione nel momento in cui si tenta di aggiungere un elemento alla struttura piena. Si tratta di una debolezza molto significativa, perché, se l'utilizzatore non ha una conoscenza certa della dimensione massima che verrà raggiunta da un contenitore, c'è il rischio di creare un array troppo grande, con la conseguenza di un'inefficiente spreco di memoria, oppure di creare un array troppo piccolo, generando un errore disastroso quando tale capacità si esaurirà.

La classe `ArrayList` della libreria di Java realizza un'astrazione più robusta, consentendo all'utilizzatore di aggiungere sempre elementi alla lista, senza che apparentemente ci sia un limite alla sua capacità totale. Per realizzare questa astrazione, Java si basa su un "trucco" che prende il nome di *array dinamico*.

In effetti, gli elementi di un esemplare di `ArrayList` sono memorizzati in un normale array, la cui esatta dimensione deve essere dichiarata all'interno del codice perché il sistema lo possa creare, assegnando come al solito a tale struttura un insieme di celle di memoria consecutive. Ad esempio, la Figura 7.2 mostra un array di 12 celle che, in un sistema di calcolo, potrebbero essere memorizzate in corrispondenza degli indirizzi di memoria che vanno da 2146 a 2157.



Figura 7.2: Un array di 12 celle, assegnate agli indirizzi di memoria che vanno da 2146 a 2157.

Dato che il sistema operativo può destinare le posizioni di memoria adiacenti alla memorizzazione di altri dati, la capacità di un array non può essere aumentata espandendolo in modo che occupi le celle vicine.

Il primo punto chiave che consente di realizzare la semantica di un array privo di limiti è che un esemplare di lista con indice gestisce al proprio interno un array che spesso ha una capacità maggiore della lunghezza della lista. Ad esempio, un utilizzatore di una lista potrebbe averla creata specificando una capacità di cinque elementi, ma il sistema potrebbe aver creato un array, interno alla lista, capace di memorizzare otto riferimenti a oggetti (invece che soltanto cinque). Questa capacità aggiuntiva rende semplice aggiungere un nuovo elemento alla fine della lista, usando la successiva cella disponibile dell'array.

Se l'utilizzatore continua ad aggiungere elementi alla lista, la capacità assegnata all'array interno prima o poi si esaurirà. In tal caso, la classe può richiedere al sistema un nuovo array, più grande, e copiare tutti i riferimenti contenuti nella lista dal vecchio array, più piccolo, alla porzione iniziale del nuovo array. A quel punto, il vecchio array non serve più e può essere riconsegnato al sistema (che, in Java, lo recupererà attraverso il garbage collector). Intuitivamente, questa strategia è molto simile a quella del paguro, che si sposta in una conchiglia più grande quando è cresciuto troppo per quella che sta abitando.

7.2.2 Implementare un array dinamico

Vediamo ora come si possa trasformare la nostra versione originale di `ArrayList`, descritta nel Codice 7.2 e 7.3, in modo che sia implementata mediante un array dinamico, con capacità non limitata. Ci basiamo sulla stessa rappresentazione interna dei dati, con un array tradizionale, A , che viene inizializzato a una capacità predefinita oppure a quella specificata come parametro del costruttore.

Il punto chiave sta nella possibilità che l'array A "cresca" di dimensione quando serve più spazio. Come già detto, non possiamo ovviamente far crescere proprio quell'array, perché la sua capacità è fissa; invece, quando l'invocazione che chiede di aggiungere un numero elemento rischia di superare la capacità dell'array che si sta utilizzando, eseguiamo le operazioni aggiuntive qui elencate:

1. Creiamo un nuovo array, B , di dimensione maggiore.
2. Assegniamo $B[k] = A[k]$ per $k = 0, 1, \dots, n - 1$, dove n è il numero di elementi della lista.
3. Assegniamo $A = B$, cioè, da questo punto in poi, usiamo il nuovo array come supporto di memorizzazione per la lista.
4. Inseriamo il nuovo elemento nel nuovo array.

L'intera procedura è illustrata nella Figura 7.3.

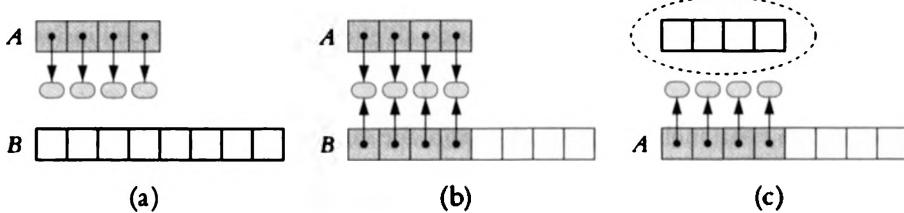


Figura 7.3: Come "cresce" un array dinamico: (a) creiamo un nuovo array, B ; (b) copiamo nella parte iniziale di B gli elementi di A ; (c) assegniamo il riferimento al nuovo array alla variabile A . Non è evidenziato il recupero, da parte del garbage collector, del vecchio array, né l'inserimento di un nuovo elemento (azione che ha provocato il ridimensionamento dell'array).

Il Codice 7.4 mostra un'implementazione concreta del metodo `resize`, che va aggiunto come metodo `protected` alla classe `ArrayList` originaria. La variabile di esemplare `data` corrisponde all'array A citato nella discussione precedente, mentre la variabile locale `temp` corrisponde a B .

Codice 7.4: Un'implementazione del metodo `ArrayList.resize`.

```
/** Ridimensiona l'array interno in modo che abbia capacità >= size. */
protected void resize(int capacity) {
    E[] temp = (E[]) new Object[capacity]; // cast sicuro; warning del compilatore
    for (int k=0; k < size; k++)
        temp[k] = data[k];
    data = temp;                                // inizia a usare il nuovo array
}
```

Il problema di cui dobbiamo ancora discutere è la dimensione del nuovo array. Spesso si usa questa regola: il nuovo array ha una capacità doppia di quella dell'array esistente, che si è riempito. Nel Paragrafo 7.2.3 presenteremo un'analisi matematica che giustifica questa scelta.

Per completare la modifica della nostra implementazione originale di `ArrayList`, riprogettiamo il metodo `add` in modo che, nel momento in cui scopre che l'array in uso è pieno, invece di lanciare un'eccezione invochi il nuovo metodo ausiliario `resize`. La versione modificata è riportata nel Codice 7.5.

Codice 7.5: Una versione modificata del metodo `ArrayList.add` (la cui versione originale si trova nel Codice 7.3) che, quando serve una capacità maggiore, invoca il metodo `resize` presentato nel Codice 7.4.

```
28  /** Inserisce e come elemento di indice i, spostando gli elementi successivi. */
29  public void add(int i, E e) throws IndexOutOfBoundsException {
30      checkIndex(i, size + 1);
31      if (size == data.length)           // non c'è spazio sufficiente
32          resize(2 * data.length);     // quindi raddoppia la capacità attuale
// il resto del metodo non viene modificato
```

Infine, osserviamo che la nostra implementazione originale della classe `ArrayList` dispone di due costruttori: un costruttore privo di parametri che usa il valore 16 come capacità iniziale e un costruttore che riceve come parametro la capacità richiesta dall'utilizzatore della lista. Con l'uso di array dinamici, tale capacità non è più un limite fisso, ma, comunque, si ottiene un'efficienza decisamente migliore in tutti quei casi in cui l'utilizzatore imposta una capacità iniziale che corrisponde all'effettiva dimensione dell'insieme dei dati che verranno inseriti nella struttura, perché questo può evitare il dispendio di tempo dovuto alla creazione degli array di dimensione intermedia e al relativo trasferimento dei dati, così come evita anche lo spreco di spazio che sarebbe conseguenza della creazione di un array troppo grande.

7.2.3 Analisi ammortizzata degli array dinamici

In questo paragrafo eseguiremo un'analisi dettagliata del tempo d'esecuzione delle operazioni che riguardano gli array dinamici. Come abbreviazione, parleremo di operazione *push* per indicare l'inserimento in una lista con indice di un elemento che diventi l'ultimo.

A prima vista la strategia che prevede di sostituire un array con uno nuovo e più grande può sembrare lenta, perché una singola operazione *push* può richiedere un tempo $\Omega(n)$ per essere eseguita, dove n è il numero di elementi presenti nell'array (e, quindi, nella

lista). Ricordiamo, dal Paragrafo 4.3.1, che la notazione Omega-grande descrive un limite inferiore asintotico per il tempo d'esecuzione di un algoritmo. Tuttavia, grazie al raddoppio della capacità durante la sostituzione dell'array, il nostro nuovo array ci consentirà di inserire n nuovi elementi prima di riempirsi. In questo modo, per ogni operazione *push* onerosa (dal punto di vista del tempo d'esecuzione) ce ne saranno molte eseguite velocemente (si veda la Figura 7.4). Questa osservazione ci consentirà di dimostrare che una sequenza di operazioni *push* eseguite su un array dinamico inizialmente vuoto è efficiente, in termini di tempo d'esecuzione complessivo.

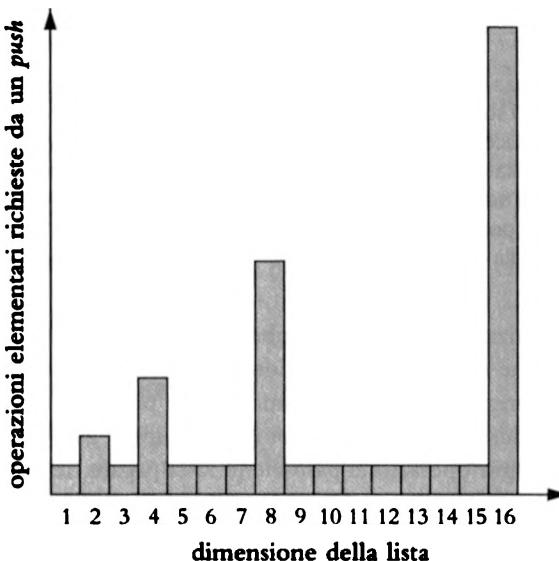


Figura 7.4: Tempo d'esecuzione di una sequenza di operazioni *push* eseguite su un array dinamico.

Usando uno schema di analisi di algoritmi chiamato *ammortamento*, dimostreremo che l'esecuzione di una sequenza di operazioni *push* su un array dinamico è, in effetti, piuttosto efficiente. Per eseguire un'*analisi ammortizzata*, usiamo una tecnica che prevede di considerare il computer come una sorta di elettrodomestico a gettone, con il quale serve il pagamento di un *ciber-dollar* per ottenere una quantità costante di tempo di elaborazione. Quando dobbiamo eseguire un'operazione, nel nostro "conto bancario" deve essere presente una quantità di ciber-dollar sufficiente a pagare per il tempo di elaborazione richiesto per quella operazione. In questo modo, la quantità totale di ciber-dollar spesi per qualunque elaborazione sarà proporzionale al tempo complessivo richiesto da essa. L'aspetto interessante di questa metodologia di analisi è che durante alcune operazioni possiamo accantonare più soldi del dovuto, per poterlo spendere nel pagamento di altre operazioni.

Proposizione 7.2: Sia L una lista con indice inizialmente vuota, con capacità unitaria, implementata mediante un array dinamico che raddoppia la propria dimensione quando è pieno. Il tempo totale speso per eseguire una sequenza di n operazioni *push* in L è $O(n)$.

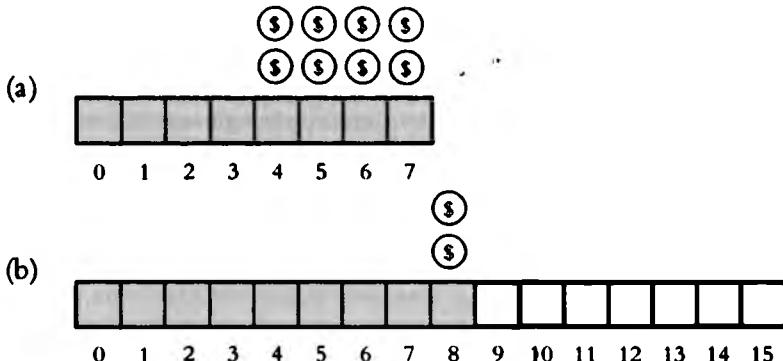


Figura 7.5: Una sequenza di operazioni *push* eseguite su un array dinamico: (a) l'array, avente 8 celle, è pieno e ci sono due ciber-dollar "accantonati" in ciascuna cella corrispondente agli indici da 4 a 7; (b) la successiva operazione *push* provoca un *overflow* e la capacità dell'array viene raddoppiata. La copiatura degli otto elementi nel nuovo array viene pagata con i ciber-dollar già presenti nell'array. L'inserimento del nuovo elemento è pagato da uno dei tre ciber-dollar imputati all'operazione *push* in corso, mentre i due ciber-dollar in eccesso vengono "accantonati" nella cella di indice 8.

Dimostrazione: Ipotizziamo che un solo ciber-dollar sia sufficiente per pagare il tempo necessario all'esecuzione di ciascuna operazione *push* eseguita in L , escludendo il tempo speso per far crescere l'array. Ancora, ipotizziamo che per portare la dimensione dell'array da k a $2k$ si debbano spendere k ciber-dollar, per il tempo impiegato per copiare gli elementi nel nuovo array. Ora, ogni operazione *push* viene fatta pagare tre ciber-dollar. In questo modo, aumentiamo artificialmente il costo di ciascuna operazione *push*, che provoca un accantonamento netto di due ciber-dollar (spesi dall'algoritmo ma non utilizzati come tempo di elaborazione): possiamo immaginare che questi due ciber-dollar, in qualche modo "risparmiati" perché l'inserimento (dovuto al *push*) non ha fatto crescere l'array, vengano virtualmente "memorizzati" all'interno della cella dell'array in cui è stato inserito l'elemento, come si può vedere nella Figura 7.5(a). La capacità dell'array viene superata (con un fenomeno di *overflow* o trabocco) quando si esegue un *push* e la lista L contiene 2^i elementi, per un qualche valore intero $i \geq 0$: in tal caso, l'array usato per rappresentare L ha una dimensione pari a 2^i . Di conseguenza, per le ipotesi fatte, il raddoppio della dimensione dell'array richiede 2^i ciber-dollar: fortunatamente, questa somma di denaro si trova già memorizzata nelle celle aventi indice che va da 2^{i-1} a $2^i - 1$ (si veda, di nuovo, la Figura 7.5).

Si noti che il precedente overflow si era verificato quando il numero di elementi era diventato per la prima volta maggiore di 2^{i-1} , per cui certamente i ciber-dollar accantonati nelle celle aventi indice che va da 2^{i-1} a $2^i - 1$ non sono ancora stati spesi. Quindi, abbiamo individuato un efficace schema di ammortamento dei costi, nel quale ciascuna operazione viene fatta pagare tre ciber-dollar e si trovano i soldi per pagare tutto il tempo di elaborazione necessario all'esecuzione dell'intera sequenza di operazioni: in effetti, paghiamo complessivamente $3n$ ciber-dollar per eseguire n operazioni *push*. In altre parole, il tempo d'esecuzione ammortizzato (cioè, in qualche modo, "suddiviso equamente") di ciascuna operazione *push* è $O(1)$ e, quindi, il tempo totale richiesto per eseguire n operazioni *push* è $O(n)$. ■

Aumento geometrico della capacità

Nonostante la dimostrazione della Proposizione 7.2 si basi sul fatto che la dimensione dell'array raddoppia ogni volta che viene aumentata, il calcolo che porta a un tempo d'esecuzione ammortizzato $O(1)$ per ciascuna operazione può essere ripetuto per qualsiasi progressione *geometrica* delle dimensioni dell'array (si veda il Paragrafo 2.2.3 per una discussione sulle progressioni geometriche). Quando si sceglie la base per la progressione geometrica, bisogna valutare il compromesso tra l'efficienza in termini di tempo d'esecuzione e di utilizzo della memoria. Se l'ultimo inserimento effettuato ha provocato un evento di ridimensionamento dell'array, con base della progressione uguale a 2 (cioè la dimensione dell'array è stata raddoppiata), in pratica l'array ha una dimensione circa doppia di quella che sarebbe necessaria. Se, invece, avessimo aumentato la dimensione dell'array soltanto del 25% della sua dimensione (usando, cioè, la base 1.25), non correremmo il rischio di sprecare così tanto spazio al termine degli inserimenti, ma avremmo speso più tempo nella fase intermedia, per effettuare un maggior numero di ridimensionamenti. Allo stesso modo è possibile dimostrare la validità delle prestazioni $O(1)$ con ammortamento usando un fattore costante maggiore dei 3 ciber-dollari per operazione che abbiamo usato nella dimostrazione della Proposizione 7.2 (come si vedrà nell'Esercizio R-7.7). Il fattore chiave per le prestazioni è che la quantità di spazio aggiunto durante ogni ridimensionamento sia proporzionale alla dimensione che ha l'array in quel momento.

Attenzione alla progressione aritmetica

Per evitare di occupare troppo spazio tutto in una volta, si potrebbe cedere alla tentazione di implementare un array dinamico con una strategia in cui ogni ridimensionamento viene effettuato aggiungendo un numero costante di celle, cioè sommando un numero costante alla dimensione, anziché moltiplicarla. Sfortunatamente, le prestazioni complessive di una tale strategia sono significativamente peggiori. Al limite, se l'aumento fosse di una sola cella, si avrebbe un ridimensionamento dell'array conseguente a ciascuna operazione *push*, dando luogo, per il calcolo del costo totale, alla ben nota sommatoria $1 + 2 + 3 + \dots + n$, che è $\Omega(n^2)$. Usando costanti di incremento della dimensione uguali a 2 o 3, la situazione migliora leggermente, come si può vedere nella Figura 7.6, ma il costo complessivo rimane quadratico.

Usando a ogni ridimensionamento un incremento fisso, generando quindi le dimensioni dell'array in progressione aritmetica, si ottiene un tempo complessivo quadratico in funzione del numero di operazioni, come enunciato dalla proposizione seguente. In effetti, quando l'insieme dei dati assume grandi dimensioni, anche un aumento di 10000 celle per ogni ridimensionamento diventerà insignificante.

Proposizione 7.3: *L'esecuzione di una sequenza di n operazioni *push* su una lista inizialmente vuota realizzata con un array dinamico che usa un incremento fisso della propria dimensione a ogni ridimensionamento richiede un tempo complessivo $\Omega(n^2)$.*

Dimostrazione: Sia $c > 0$ l'incremento costante della capacità dell'array che avviene in seguito a ogni evento di ridimensionamento. Durante l'esecuzione di n operazioni *push* in sequenza, si impiegherà tempo per trasferire gli elementi in array di dimensione $c, 2c, 3c, \dots, mc$, con $m = \lceil n/c \rceil$ e, quindi, il tempo totale è proporzionale a $c + 2c + 3c + \dots + mc$.

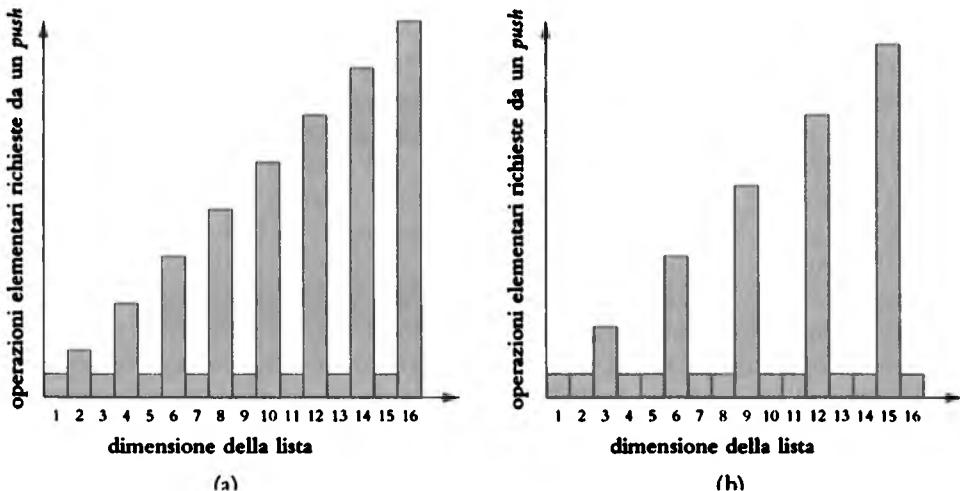


Figura 7.6: Tempo d'esecuzione di una sequenza di operazioni *push* eseguite su un array dinamico usando dimensioni dell'array in progressione aritmetica. La figura (a) ipotizza un incremento di 2 unità a ogni ridimensionamento, mentre la figura (b) usa la costante 3.

Per la Proposizione 4.3, questa somma è:

$$\sum_{i=1}^m c_i = c \cdot \sum_{i=1}^m i = c \frac{m(m+1)}{2} \geq c \frac{\frac{n}{c} \left(\frac{n}{c} + 1 \right)}{2} \geq \frac{1}{2c} \cdot n^2$$

Quindi, l'esecuzione di n operazioni *push* richiede un tempo $\Omega(n^2)$. ■

Utilizzo della memoria e diminuzione della dimensione dell'array

Un'altra conseguenza della regola di aumento geometrico della capacità di un array dinamico quando vi si aggiungono elementi è che la dimensione finale dell'array è certamente proporzionale al numero complessivo di elementi inseriti nella lista, cioè la struttura dati usa una quantità di memoria $O(n)$: una proprietà molto interessante.

Se un contenitore, come una lista con indice, mette a disposizione dell'utilizzatore anche operazioni che provocano la rimozione di elementi, occorre fare maggiore attenzione perché si possa garantire che l'array dinamico utilizzato per memorizzarne gli elementi usi una quantità di memoria $O(n)$. Il rischio è che ripetuti inserimenti possano far crescere l'array e che, poi, dopo aver rimosso molti elementi, la dimensione dell'array non sia più proporzionale al numero di elementi rimasti nella lista (e nell'array).

Un'implementazione robusta di questa struttura dati ridurrà la dimensione dell'array dinamico interno quando necessario, pur mantenendo un costo ammortizzato $O(1)$ per le singole operazioni. Bisogna, però, fare molta attenzione a non provocare continue oscillazioni della dimensione dell'array, per effetto di aumenti e diminuzioni che si alternano, nel qual caso il limite ammortizzato non verrà preservato. Nell'Esercizio C-7.29 analizzeremo una strategia che prevede di dimezzare la dimensione dell'array ogni volta che il numero effettivo di elementi memorizzati scende al di sotto di un quarto della capacità, garantendo così che tale capacità sia sempre al massimo il quadruplo del nume-

ro di elementi; l'analisi ammortizzata di tale strategia sarà oggetto degli Esercizi C-7.30 e C-7.31.

7.2.4 La classe `StringBuilder` di Java

Nelle prime pagine del Capitolo 4 abbiamo descritto un esperimento nel quale confrontavamo due algoritmi progettati per generare una stringa di grandi dimensioni (Codice 4.2). Il primo si basava sulla concatenazione ripetuta, usando la classe `String`, mentre il secondo usava la classe `StringBuilder`. Abbiamo osservato che la seconda versione era significativamente più veloce, con l'evidenza sperimentale che suggeriva un tempo d'esecuzione quadratico per il primo algoritmo e un tempo lineare per il secondo. Ora siamo in grado di spiegare le motivazioni teoriche che stanno alla base di quelle osservazioni.

La classe `StringBuilder` rappresenta una stringa modificabile e ne memorizza i caratteri in un array dinamico. Con analisi simile a quella vista nella Proposizione 7.2, questa implementazione garantisce che una sequenza di operazioni di "aggiunta in fondo" (*append*) che produca una stringa di lunghezza n viene eseguita in un tempo complessivo $O(n)$ (gli inserimenti in posizioni diverse dalla fine della stringa non portano a questa garanzia, esattamente come per un esemplare di `ArrayList`).

L'utilizzo ripetuto della concatenazione tra stringhe richiede, invece, un tempo quadratico, come abbiamo già visto con l'analisi eseguita nel Paragrafo 4.3.3. In effetti, questo approccio è simile a un array dinamico con una progressione aritmetica della dimensione su base unitaria, copiando ripetutamente tutti i caratteri da un array a un nuovo array la cui dimensione è aumentata di una sola unità.

7.3 Liste posizionali

Quando si lavora con sequenze basate su array, l'uso di numeri interi come indici è un metodo eccellente per descrivere la posizione di un elemento o la posizione in cui deve avvenire un inserimento o una rimozione. Al contrario, gli indici numerici non sono una buona scelta quando si vogliono descrivere le posizioni all'interno di una lista concatenata, perché, conoscendo soltanto l'indice di un elemento, l'unico modo per raggiungerlo è la scansione della lista, un elemento dopo l'altro, partendo dall'inizio o dalla fine, contando gli elementi mentre si procede.

Inoltre, gli indici non sono un'astrazione comoda nemmeno per avere una visione più locale della posizione all'interno di una sequenza, perché l'indice di un elemento cambia nel tempo per effetto di inserimenti o rimozioni che avvengono in punti precedenti della sequenza. Ad esempio, non è particolarmente comodo descrivere la posizione di una persona in attesa in coda basandosi su un indice, perché per farlo è necessario sapere, istante per istante, quale sia la distanza della persona dall'inizio della coda. È preferibile utilizzare un'astrazione, come nella Figura 7.7, che descriva una posizione in qualche altro modo.

Il nostro obiettivo, quindi, è la progettazione di un tipo di dato astratto che fornisca all'utilizzatore della struttura una modalità per fare riferimento agli elementi in qualsiasi posizione della sequenza, con la possibilità di eseguire inserimenti e rimozioni in qualsiasi

punto. Questo ci consentirà di descrivere in modo efficiente situazioni come quella di una persona che decide di abbandonare una coda prima di raggiungerne la posizione iniziale, oppure di un persona che “salta” dentro alla coda subito prima o subito dopo un amico.

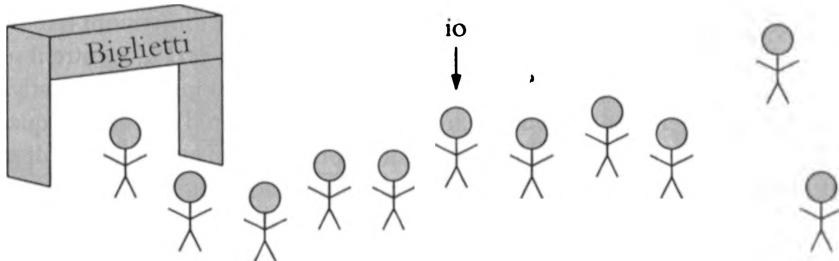


Figura 7.7 Vogliamo poter identificare la posizione di un elemento in una sequenza senza usare un numero intero come indice. L'etichetta "io" rappresenta una qualche astrazione che identifichi la posizione.

Come ulteriore esempio, possiamo immaginare un documento di testo come se fosse una lunga sequenza di caratteri. Un elaboratore di testi (*word processor*) utilizza l'astrazione del *cursor* per descrivere una posizione all'interno del documento, senza usare esplicitamente un numero intero come indice, consentendo così operazioni come “cancella il carattere su cui è posizionato il cursore” oppure “inserisci un nuovo carattere subito dopo il cursore”. Inoltre, in questo modo possiamo essere in grado di fare riferimento a una posizione relativa alla struttura del documento, come l'inizio di uno specifico capitolo, senza doverci basare sull'indice di un carattere (o sul numero di un capitolo), che può cambiare mentre modifichiamo il documento stesso.

Per tutti questi motivi, mettiamo da parte per un po' i metodi dell'interfaccia `List` di Java che si basano su indici e, invece, sviluppiamo un tipo di dato astratto che chiameremo *lista posizionale* (*positional list*). Anche se si tratta di un'astrazione e non è necessario che si basi su una implementazione mediante lista concatenata, mentre progettiamo questo ADT avremo certamente in mente una lista concatenata e cercheremo di sfruttare al meglio le caratteristiche peculiari di una lista concatenata, come il fatto di poter inserire e rimuovere elementi in qualunque posizione in un tempo $O(1)$, cosa che non è possibile con una sequenza basata su array.

Nel progettare questo ADT ci scontriamo subito con un primo problema: per poter effettuare inserimenti e rimozioni in qualunque posizione in un tempo costante, abbiamo bisogno di una riferimento al nodo in cui l'elemento è memorizzato. Di conseguenza, siamo fortemente tentati di progettare un tipo di dato astratto in cui il riferimento a un nodo serva come meccanismo per descrivere la posizione di un elemento. In effetti, la nostra classe `DoublyLinkedList`, vista nel Paragrafo 3.4.1, dispone dei metodi `addBetween` e `remove` che accettano riferimenti ai nodi come parametri, anche se abbiamo volutamente dichiarato privati quei metodi.

Purtroppo, l'utilizzo pubblico dei nodi nella definizione dell'ADT violerebbe due principi della progettazione orientata agli oggetti, astrazione e encapsulamento, così come

sono stati presentati nel Capitolo 2. Sono molti i motivi che ci hanno fatto preferire l'incapsulamento dei nodi di una lista concatenata, tanto per la sicurezza del nostro codice quanto per i benefici derivanti agli utilizzatori della nostra astrazione:

- Gli utilizzatori della nostra struttura dati saranno agevolati dal fatto di non doversi preoccupare dei dettagli della nostra implementazione, come la manipolazione dei nodi a basso livello o l'utilizzo dei nodi sentinella. Si noti, ad esempio, che per poter utilizzare il metodo `addBetween` della classe `DoublyLinkedList` per aggiungere un nodo all'inizio della sequenza, bisogna passare come parametro la sentinella iniziale.
- Se non permettiamo agli utilizzatori di accedere ai nodi o manipolarli, possiamo rendere più robusta la struttura dati, perché siamo in grado di garantire che nessuna azione violerà la coerenza della lista, dato che i collegamenti tra i nodi non possono essere gestiti in modo errato. Se consentissimo l'invocazione diretta dei metodi `addBetween` o `remove` della classe `DoublyLinkedList`, potrebbe verificarsi un problema piuttosto subdolo nei casi in cui venisse passato come parametro un nodo che non appartiene alla lista (tornate a rivedere il codice di questi metodi e scoprirete cosa provoca questo problema!).
- Incapsulando in modo migliore i dettagli interni della nostra implementazione, ottieniamo una maggiore flessibilità nel momento in cui vorremo modificare il progetto della struttura dati, per migliorarne le prestazioni. In effetti, con un'astrazione ben progettata, potremmo proporre il concetto di posizione non numerica anche usando una sequenza basata su array (come nell'Esercizio C-7.43).

Per concludere, per definire il tipo di dato astratto "lista posizionale" introduciamo il concetto di *posizione*, che formalizza il principio intuitivo di "posizione" di un elemento relativamente agli altri all'interno della lista (nel momento in cui useremo una lista concatenata per implementare questo ADT, vedremo come i riferimenti ai nodi potranno essere usati, in modalità privata, come espressione naturale delle posizioni).

7.3.1 Posizioni

Come generale astrazione della collocazione di un elemento all'interno di una struttura, definiamo un semplice tipo di dato astratto, che chiamiamo *posizione* (*position*). Una posizione mette a disposizione soltanto il metodo seguente:

`getElement()`: Restituisce l'elemento memorizzato in questa posizione.

Una posizione funge da marcatore all'interno di una lista posizionale. La posizione p , associata a un elemento e della lista L , non cambia anche se l'indice di e in L si modifica per effetto di inserimenti o rimozioni che avvengono nella lista. Inoltre, la posizione p non cambia nemmeno se sostituiamo l'elemento e memorizzato in p con un altro elemento. L'unica azione che rende non più valida una posizione è una rimozione esplicita dalla lista che coinvolga la posizione stessa (e il suo elemento).

La definizione formale del tipo "posizione" consente il suo utilizzo come parametro di metodi e come valore restituito da altri metodi del tipo di dato astratto "lista posizionale", come stiamo per descrivere.

7.3.2 Il tipo di dato astratto "lista posizionale"

A questo punto possiamo vedere una *lista posizionale* come un contenitore di posizioni, ciascuna delle quali memorizza un elemento. I metodi di accesso messi a disposizione dall'ADT lista posizionale comprendono i seguenti (enunciati per la lista L):

- first():** Restituisce la posizione del primo elemento di L (o `null` se la lista è vuota).
- last():** Restituisce la posizione dell'ultimo elemento di L (o `null` se la lista è vuota).
- before(p):** Restituisce la posizione di L che precede immediatamente la posizione p (o `null` se p è la prima posizione).
- after(p):** Restituisce la posizione di L che segue immediatamente la posizione p (o `null` se p è l'ultima posizione).
- isEmpty():** Restituisce `true` se e solo se la lista L non contiene elementi.
- size():** Restituisce il numero di elementi presenti nella lista L .

Se la posizione p , passata come parametro a un metodo, non è una posizione valida per la lista L , si verifica un errore.

Si osservi che i metodi `first()` e `last()` dell'ADT lista posizionale restituiscono le *posizioni* corrispondenti, non gli *elementi* (diversamente da quanto avviene per gli omonimi metodi dell'ADT coda doppia). Il primo elemento di una lista posizionale si può determinare invocando, poi, il metodo `getElement` con la posizione ottenuta dal metodo `first`, in questo modo: `first().getElement()`. Ricevendo una posizione come valore restituito, c'è il vantaggio di poterla, poi, utilizzare per spostarsi all'interno della lista.

Come esempio di tipica scansione o attraversamento di una lista posizionale, vediamo il Codice 7.6, che scandisce la lista `guests` che contiene stringhe come elementi e le visualizza una dopo l'altra mentre procede dall'inizio alla fine della lista.

Codice 7.6: Scansione di una lista posizionale.

```

1 Position<String> cursor = guests.first();
2 while (cursor != null) {
3     System.out.println(cursor.getElement());
4     cursor = guests.after(cursor); // sposta il cursore alla posizione successiva
5 }                                // se questa esiste

```

Questo codice si basa sulla convenzione che il metodo `after` restituisca il riferimento nullo quando viene invocato passando l'ultima posizione della lista (tale valore restituito è chiaramente distinguibile da qualunque posizione valida). La definizione dell'ADT lista posizionale indica, analogamente, che il valore nullo sarà restituito quando il metodo `before` viene invocato passando la prima posizione come parametro, oppure quando i metodi `first` o `last` vengono invocati con una lista vuota. Il codice qui presentato, quindi, funziona correttamente anche se la lista `guests` è vuota.

Metodi di aggiornamento di una lista posizionale

Il tipo di dato astratto “lista posizionale” definisce anche i seguenti metodi di aggiornamento:

- addFirst(*e*)**: Inserisce il nuovo elemento *e* all'inizio della lista, restituendo la sua posizione.
- addLast(*e*)**: Inserisce il nuovo elemento *e* alla fine della lista, restituendo la sua posizione.
- addBefore(*p*, *e*)**: Inserisce il nuovo elemento *e* nella lista, immediatamente prima della posizione *p*, restituendo la posizione del nuovo elemento.
- addAfter(*p*, *e*)**: Inserisce il nuovo elemento *e* nella lista, immediatamente dopo la posizione *p*, restituendo la posizione del nuovo elemento.
- set(*p*, *e*)**: Sostituisce l'elemento in posizione *p* con l'elemento *e*, restituendo l'elemento che si trovava precedentemente in *p*.
- remove(*p*)**: Elimina e restituisce l'elemento che si trova in posizione *p* nella lista, rendendo poi non più valida tale posizione.

Ad un primo esame sembra che nell'elenco di operazioni messe a disposizione da questo ADT ci sia ridondanza, perché per eseguire l'operazione **addFirst(*e*)** potremmo scrivere **addBefore(first(), *e*)**, così come lo stesso risultato dell'operazione **addLast(*e*)** si potrebbe ottenere con la combinazione **addAfter(last(), *e*)**: queste invocazioni sostitutive, però, funzionerebbero soltanto nel caso di lista non vuota.

Esempio 7.4: La tabella seguente mostra una sequenza di operazioni eseguite su una lista posizionale, inizialmente vuota, che può memorizzare numeri interi. Per identificare le posizioni usiamo variabili come *p* e *q*. Per rendere più agevole la presentazione dei dati, nella visualizzazione del contenuto della lista usiamo un pedice per indicare la posizione in cui è memorizzato ciascun particolare elemento.

Operazione	Valore restituito	Contenuto della lista
addLast(8)	<i>p</i>	(8 _{<i>p</i>})
first()	<i>p</i>	(8 _{<i>p</i>})
addAfter(<i>p</i>, 5)	<i>q</i>	(8 _{<i>p</i>} , 5 _{<i>q</i>})
before(<i>q</i>)	<i>p</i>	(8 _{<i>p</i>} , 5 _{<i>q</i>})
addBefore(<i>q</i>, 3)	<i>r</i>	(8 _{<i>p</i>} , 3 _{<i>r</i>} , 5 _{<i>q</i>})
<i>r</i>.getElement()	3	(8 _{<i>p</i>} , 3 _{<i>r</i>} , 5 _{<i>q</i>})
after(<i>p</i>)	<i>r</i>	(8 _{<i>p</i>} , 3 _{<i>r</i>} , 5 _{<i>q</i>})
before(<i>p</i>)	null	(8 _{<i>p</i>} , 3 _{<i>r</i>} , 5 _{<i>q</i>})
addFirst(9)	<i>s</i>	(9 _{<i>s</i>} , 8 _{<i>p</i>} , 3 _{<i>r</i>} , 5 _{<i>q</i>})
remove(last())	5	(9 _{<i>s</i>} , 8 _{<i>p</i>} , 3 _{<i>r</i>})
set(<i>p</i>, 7)	8	(9 _{<i>s</i>} , 7 _{<i>p</i>} , 3 _{<i>r</i>})
remove(<i>q</i>)	“error”	(9 _{<i>s</i>} , 7 _{<i>p</i>} , 3 _{<i>r</i>})

Definizione delle interfacce in Java

Siamo ormai pronti per formalizzare le definizioni dei tipi di dati astratti "posizione" e "lista posizionale". Il Codice 7.7 riporta l'interfaccia Position in Java. Nel seguito, il Codice 7.8 presenta la definizione, sempre in Java, della nostra interfaccia PositionalList. Se il metodo `getElement()` viene invocato con un esemplare di Position che è stato precedentemente eliminato dalla lista a cui apparteneva, verrà lanciata un'eccezione di tipo `IllegalStateException`, mentre se a un metodo di un esemplare di PositionalList viene fornito come parametro un esemplare di Position non valido, verrà lanciata un'eccezione di tipo `IllegalArgumentException` (entrambe queste eccezioni sono definite nella libreria standard di Java).

Codice 7.7: L'interfaccia Position.

```

1 public interface Position<E> {
2     /**
3      * Restituisce l'elemento memorizzato in questa posizione.
4      *
5      * @return l'elemento memorizzato
6      * @throws IllegalStateException se la posizione non è più valida
7      */
8     E getElement() throws IllegalStateException;
9 }
```

Codice 7.8: L'interfaccia PositionalList.

```

1 /** L'interfaccia che definisce liste posizionali. */
2 public interface PositionalList<E> {
3
4     /** Restituisce il numero di elementi presenti nella lista. */
5     int size();
6
7     /** Restituisce true se e solo se la lista è vuota. */
8     boolean isEmpty();
9
10    /** Restituisce la prima Position della lista (o null se la lista è vuota). */
11    Position<E> first();
12
13    /** Restituisce l'ultima Position della lista (o null se la lista è vuota). */
14    Position<E> last();
15
16    /** Restituisce la Position che precede p (o null, se p è la prima). */
17    Position<E> before(Position<E> p) throws IllegalArgumentException;
18
19    /** Restituisce la Position che segue p (o null, se p è l'ultima). */
20    Position<E> after(Position<E> p) throws IllegalArgumentException;
21
22    /** Inserisce l'elemento e all'inizio della lista; ne restituisce la posizione. */
23    Position<E> addFirst(E e);
24
25    /** Inserisce l'elemento e alla fine della lista; ne restituisce la posizione. */
26    Position<E> addLast(E e);
27
28    /** Inserisce l'elemento e prima della Position p; ne restituisce la posizione. */
29    Position<E> addBefore(Position<E> p, E e)
30        throws IllegalArgumentException;
31 }
```

```

32  /** Inserisce l'elemento e dopo la Position p; ne restituisce la posizione. */
33  Position<E> addAfter(Position<E> p, E e)
34      throws IllegalArgumentException;
35
36  /** Sostituisce l'elemento nella Position p; restituisce l'elemento sostituito. */
37  E set(Position<E> p, E e) throws IllegalArgumentException;
38
39  /** Elimina e restituisce l'elemento nella Position p (poi p non è più valida). */
40  E remove(Position<E> p) throws IllegalArgumentException;
41 }

```

7.3.3 Implementazione con lista doppiamente concatenata

Probabilmente non sarà una sorpresa: sceglieremo di implementare l’interfaccia `PositionalList` mediante una lista doppiamente concatenata (anche se abbiamo già realizzato la classe `DoublyLinkedList`, nel Capitolo 3, non si tratta di un’implementazione dell’interfaccia `Positionalist`, perché non ne rispetta le specifiche).

In questo paragrafo svilupperemo un’implementazione concreta dell’interfaccia `Positionalist` usando una lista doppiamente concatenata. I dettagli di basso livello della nostra nuova rappresentazione di lista concatenata, come l’uso delle sentinelle iniziale e finale, saranno identici a quelli della nostra prima versione e, infatti, rimandiamo al Paragrafo 3.4 i lettori interessati a una trattazione delle operazioni tipiche di una lista doppiamente concatenata. Ciò che cambia in questo paragrafo è la gestione dell’astrazione di posizione.

Il modo più ovvio per identificare le posizioni all’interno di una lista concatenata è l’utilizzo dei riferimenti ai nodi, quindi dichiariamo la classe `Node`, annidata all’interno della nostra lista concatenata, in modo che implementi l’interfaccia `Position`, fornendo il necessario supporto al metodo `getElement`. Quindi, i nodi *sono* le posizioni. La classe `Node`, comunque, è dichiarata privata, per preservare adeguatamente l’incapsulamento. Tutti i metodi pubblici della lista posizionale sono definiti per utilizzare il tipo `Position`, per cui, anche se, come programmatore, sappiamo che stiamo trasferendo riferimenti ai nodi, all’esterno della lista tali riferimenti sono noti come semplici posizioni: di conseguenza, gli utilizzatori della nostra classe potranno invocare con tali riferimenti solamente il metodo `getElement()`.

Nel Codice 7.9, 7.10, 7.11 e 7.12 definiamo la classe `LinkedPositionalist`, che implementa il tipo di dato astratto “lista posizionale”. Alcuni commenti al codice:

- Il Codice 7.9 contiene la definizione della classe annidata `Node<E>`, che implementa l’interfaccia `Position<E>`. Di seguito troviamo la dichiarazione delle variabili di esemplare della classe esterna `LinkedPositionalist` e il suo costruttore.
- Il Codice 7.10 inizia con due importanti metodi ausiliari che ci aiutano a rendere robusti i cast tra i tipi `Position` e `Node`. Il metodo `validate(p)` viene invocato ogni volta che l’utilizzatore passa come parametro un esemplare di `Position`, *p*, e lancia un’eccezione se identifica la posizione come non valida, altrimenti restituisce la posizione stessa, dopo averla convertita, con un cast esplicito, in un riferimento di tipo `Node`, in modo tale che poi si possano invocare metodi della classe `Node`. Il metodo privato `position(nodo)` viene utilizzato subito prima di restituire all’utilizzatore di uno dei metodi pubblici un riferimento a un *nodo*, che deve diventare di tipo `Position`. Il suo scopo principale è quello di garantire che non venga restituito all’invocante un nodo sentinella, resti-

tuendo in tal caso, invece, il riferimento `null`. I metodi pubblici che seguono usano questi metodi ausiliari privati.

- Il Codice 7.11 contiene la maggior parte dei metodi di aggiornamento pubblici, sfruttando il metodo privato `addBetween` per rendere omogenea l'implementazione delle diverse operazioni di inserimento.
- Il Codice 7.12, infine, presenta il metodo pubblico `remove`. Si osservi che riporta al valore `null` tutti i campi del nodo che viene rimosso: una condizione che verrà poi verificata per riconoscere una posizione che non è più valida.

Codice 7.9: Un'implementazione della classe `LinkedPositionalist` (prosegue con il Codice 7.10, 7.11 e 7.12).

```

1  /** Implementazione di lista posizionale mediante lista doppiamente concatenata. */
2  public class LinkedPositionalist<E> implements Positionalist<E> {
3      //----- classe Node annidata -----
4      private static class Node<E> implements Position<E> {
5          private E element;    // riferimento all'elemento memorizzato in questo nodo
6          private Node<E> prev; // riferimento al nodo precedente nella lista
7          private Node<E> next; // riferimento al nodo seguente nella lista
8          public Node(E e, Node<E> p, Node<E> n) {
9              element = e;
10             prev = p;
11             next = n;
12         }
13         public E getElement() throws IllegalStateException {
14             if (next == null)    // identifica convenzionalmente i nodi non più validi
15                 throw new IllegalStateException("Position no longer valid");
16             return element;
17         }
18         public Node<E> getPrev() {
19             return prev;
20         }
21         public Node<E> getNext() {
22             return next;
23         }
24         public void setElement(E e) {
25             element = e;
26         }
27         public void setPrev(Node<E> p) {
28             prev = p;
29         }
30         public void setNext(Node<E> n) {
31             next = n;
32         }
33     } //----- fine della classe Node annidata -----
34
35     // variabili di esemplare della classe LinkedPositionalist
36     private Node<E> header; // sentinella iniziale
37     private Node<E> trailer; // sentinella finale
38     private int size = 0;    // numero di elementi presenti nella lista
39
40     /** Costruisce una nuova lista vuota. */
41     public LinkedPositionalist() {
42         header = new Node<E>(null, null, null); // crea la sentinella iniziale
43         trailer = new Node<E>(null, header, null); // trailer è preceduto da header

```

```

    header.setNext(trailer);           // header è seguito da trailer
}

```

Codice 7.10: Un'implementazione della classe `LinkedPositionalList` (continua dal Codice 7.9 e prosegue con il Codice 7.11 e 7.12).

```

46   // metodi ausiliari privati
47   /** Verifica se la posizione è valida e la restituisce sotto forma di nodo. */
48   private Node<E> validate(Position<E> p) throws IllegalArgumentException {
49     if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
50     Node<E> node = (Node<E>) p; // cast sicuro
51     if (node.getNext() == null) // per convenzione, è un nodo non valido
52       throw new IllegalArgumentException("p is no longer in the list");
53     return node;
54   }
55
56   /** Restituisce il nodo sotto forma di Position (o null, se è una sentinella). */
57   private Position<E> position(Node<E> node) {
58     if (node == header || node == trailer)
59       return null; // non passa le sentinelle all'utilizzatore
60     return node;
61   }
62
63   // metodi pubblici di accesso
64   /** Restituisce il numero di elementi presenti nella lista concatenata. */
65   public int size() { return size; }
66
67   /** Restituisce true se e solo se la lista concatenata è vuota. */
68   public boolean isEmpty() { return size == 0; }
69
70   /** Restituisce la prima Position della lista (o null se la lista è vuota). */
71   public Position<E> first() {
72     return position(header.getNext());
73   }
74
75   /** Restituisce l'ultima Position della lista (o null se la lista è vuota). */
76   public Position<E> last() {
77     return position(trailer.getPrev());
78   }
79
80   /** Restituisce la Position che precede p (o null, se p è la prima). */
81   public Position<E> before(Position<E> p) throws IllegalArgumentException {
82     Node<E> node = validate(p);
83     return position(node.getPrev());
84   }
85
86   /** Restituisce la Position che segue p (o null, se p è l'ultima). */
87   public Position<E> after(Position<E> p) throws IllegalArgumentException {
88     Node<E> node = validate(p);
89     return position(node.getNext());
90   }

```

Codice 7.11: Un'implementazione della classe `LinkedPositionalList` (continua dal Codice 7.9 e 7.10, e prosegue con il Codice 7.12).

```

    // metodi ausiliari privati
    /** Aggiunge l'elemento e alla lista concatenata tra i due nodi dati. */
    private Position<E> addBetween(E e, Node<E> pred, Node<E> succe) {

```

```

94     Node<E> newest = new Node<E>(e, pred, succ); // crea il nuovo nodo e lo collega
95     pred.setNext(newest);
96     succ.setPrev(newest);
97     size++;
98     return newest;
99 }
100
101 // metodi pubblici di aggiornamento
102 /** Inserisce l'elemento e all'inizio della lista; ne restituisce la posizione. */
103 public Position<E> addFirst(E e) {
104     return addBetween(e, header, header.getNext()); // subito dopo header
105 }
106
107 /** Inserisce l'elemento e alla fine della lista; ne restituisce la posizione. */
108 public Position<E> addLast(E e) {
109     return addBetween(e, trailer.getPrev(), trailer); // subito prima di trailer
110 }
111
112 /** Inserisce l'elemento e prima della Position p; ne restituisce la posizione. */
113 public Position<E> addBefore(Position<E> p, E e)
114     throws IllegalArgumentException {
115     Node<E> node = validate(p);
116     return addBetween(e, node.getPrev(), node);
117 }
118
119 /** Inserisce l'elemento e dopo la Position p; ne restituisce la posizione. */
120 public Position<E> addAfter(Position<E> p, E e)
121     throws IllegalArgumentException {
122     Node<E> node = validate(p);
123     return addBetween(e, node, node.getNext());
124 }
125
126 /** Sostituisce l'elemento nella Position p; restituisce l'elemento sostituito. */
127 public E set(Position<E> p, E e) throws IllegalArgumentException {
128     Node<E> node = validate(p);
129     E answer = node.getElement();
130     node.setElement(e);
131     return answer;
132 }

```

Codice 7.12: Un'implementazione della classe `LinkedPositionalList` (continua dal Codice 7.9, 7.10 e 7.11).

```

/** Elimina e restituisce l'elemento nella Position p (poi p non è più valida). */
public E remove(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    E answer = node.getElement();
    node.setElement(null); // per aiutare il garbage collector
    node.setNext(null); // per rispettare la convenzione sui nodi non validi
    node.setPrev(null);
    return answer;
}

```

Le prestazioni di una lista posizionale concatenata

Il tipo di dato astratto “lista posizionale” si adatta perfettamente a un’implementazione mediante lista doppiamente concatenata, perché tutte le operazioni vengono eseguite in un tempo costante nel caso peggiore, come si può vedere nella Tabella 7.2, in forte contrasto con la struttura `ArrayList` (analizzata nella Tabella 7.1), che richiede un tempo lineare per le operazioni di inserimento e rimozione che avvengano in posizioni arbitrarie, per effetto del ciclo che deve far scorrere alcuni elementi di un posto.

Ovviamente la nostra lista posizionale non consente l’utilizzo dei metodi basati sugli indici, che fanno invece parte dell’interfaccia `List` ufficiale, che abbiamo presentato nel Paragrafo 7.1. Si può aggiungere il supporto a quei metodi effettuando una scansione della lista mentre si contano i nodi (come nell’Esercizio C-7.38), ma questo richiede un tempo proporzionale alla dimensione della sottolista analizzata.

Tabella 7.2: Prestazioni di una lista posizionale avente n elementi realizzata mediante una lista doppiamente concatenata. Lo spazio utilizzato è $O(n)$.

Metodo	Tempo d'esecuzione
<code>size()</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>first(), last()</code>	$O(1)$
<code>before(<i>p</i>).after(<i>p</i>)</code>	$O(1)$
<code>addFirst(<i>e</i>), addLast(<i>e</i>)</code>	$O(1)$
<code>addBefore(<i>p</i>, <i>e</i>), addAfter(<i>p</i>, <i>e</i>)</code>	$O(1)$
<code>set(<i>p</i>, <i>e</i>)</code>	$O(1)$
<code>remove(<i>p</i>)</code>	$O(1)$

Implementazione di una lista posizionale con un array

Una lista posizionale L può essere implementata anche usando un array A per la memorizzazione dei suoi elementi, ma occorre fare un po’ di attenzione nel progetto degli oggetti che serviranno a rappresentare le posizioni. A prima vista, si potrebbe immaginare che una posizione p debba memorizzare soltanto l’indice i associato all’elemento che si trova quella posizione, elemento che poi sarà effettivamente memorizzato nella cella i dell’array: in questo modo si può implementare il metodo `getElement(p)` che restituisca semplicemente $A[i]$. Il problema di questo approccio è che l’indice associato a un elemento, e , cambia quando avvengono inserimenti o rimozioni in posizioni precedenti. Se la lista ha già restituito all’utilizzatore una posizione, p , associata all’elemento e , questa conterrà al proprio interno un indice i non aggiornato e l’uso di tale posizione porterebbe a un accesso a una cella sbagliata nell’array (non va dimenticato che in una lista posizionale le posizioni devono sempre essere definite in modo relativo rispetto alle posizioni vicine e non in funzione degli indici).

Di conseguenza, se vogliamo implementare una lista posizionale usando un array, ci serve un approccio diverso. Suggeriamo la rappresentazione seguente: invece di memorizzare gli elementi di L direttamente nell’array A , in ciascuna cella di A memorizziamo un nuovo tipo di oggetto-posizione, che a sua volta memorizza al proprio interno un elemento, e , e l’indice i associato a tale elemento nella lista, come illustrato nella Figura 7.8.

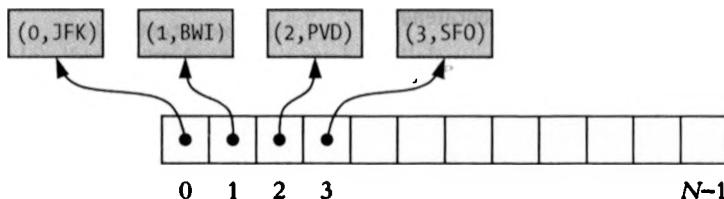


Figura 7.8: Una rappresentazione di una lista posizionale basata su array.

Con questa rappresentazione siamo in grado di determinare tanto l'indice associato a una posizione quanto la posizione associata a un indice. È possibile, quindi, implementare un metodo d'accesso, come `before(p)`, trovando l'indice memorizzato all'interno della posizione data e usando, poi, l'array per individuare le posizioni adiacenti.

Quando un elemento viene inserito o rimosso in qualche punto della lista, possiamo eseguire un ciclo che scandisca l'array per aggiornare la variabile indice memorizzata in tutte le posizioni successive della lista, che durante l'aggiornamento sono state fatte scorrere.

Efficienza di una sequenza basata su array

In questa implementazione di sequenza basata su array, i metodi `addFirst`, `addBefore`, `addAfter` e `remove` richiedono un tempo $O(n)$, perché devono far scorrere le posizioni degli oggetti per far posto a una nuova posizione o per riempire il buco creato dalla rimozione della posizione richiesta (esattamente come avviene per i metodi di inserimento e rimozione nelle implementazioni basate su indice). Tutti gli altri metodi basati sulla posizione sono $O(1)$.

7.4 Iteratori

Un **iteratore** è uno schema di progettazione software che astrae il processo di scansione di una sequenza di elementi, effettuata un elemento per volta. Gli elementi da scandire possono essere memorizzati in una classe contenitore, inviati come flusso attraverso la rete oppure, ancora, generati da una serie di elaborazioni.

Per rendere omogeneo il trattamento e la sintassi della scansione (o iterazione) di oggetti in modo che sia indipendente dall'organizzazione della sequenza, Java definisce l'interfaccia `java.util.Iterator`, dotata dei due metodi seguenti:

hasNext(): Restituisce `true` se e solo se c'è almeno un altro elemento nella sequenza.

next(): Restituisce il successivo elemento della sequenza.

L'interfaccia usa l'infrastruttura di programmazione generica di Java, per cui il metodo `next()` restituisce un elemento di tipo parametrico. Ad esempio, la classe `Scanner` (descritta nel Paragrafo 1.6) implementa formalmente l'interfaccia `Iterator<String>`, per cui il suo metodo `next()` restituisce un esemplare di `String`.

Se il metodo `next()` di un iteratore viene invocato quando non ci sono più elementi disponibili, viene lanciata un'eccezione di tipo `NoSuchElementException`, anche se, ovviamente, si può individuare tale condizione prima di invocare `next()` usando il metodo `hasNext()`.

L'utilizzo coordinato di questi due metodi consente di progettare un ciclo generico che elabori gli elementi restituiti dall'iteratore. Ad esempio, se la variabile `iter` fa riferimento a un esemplare di tipo `Iterator<String>`, possiamo scrivere:

```
while (iter.hasNext()) {
    String value = iter.next();
    System.out.println(value);
}
```

L'interfaccia `java.util.Iterator` contiene un terzo metodo, che è implementato *in modo facoltativo* da alcuni iteratori:

`remove()`: Elimina dalla raccolta l'elemento che è stato restituito dalla più recente invocazione di `next()`. Lancia una `IllegalStateException` se `next` non è mai stato invocato o se `remove` è già stato invocato dopo la più recente invocazione di `next`.

Questo metodo può essere utilizzato per eliminare alcuni selezionati elementi da una raccolta, ad esempio per rimuovere da un insieme di dati tutti e soli i numeri negativi.

Per semplicità, nella maggior parte degli iteratori di questo libro non implementeremo tale metodo `remove`, ma ne vedremo due esempi concreti più avanti, in questo stesso paragrafo. Se un iteratore non consente la rimozione di elementi, questo metodo per convenzione lancia un'eccezione di tipo `UnsupportedOperationException`.

7.4.1 L'interfaccia `Iterable` e il ciclo `for-each` in Java

Un esemplare di iteratore consente di eseguire un'unica scansione della raccolta a cui è associato: si può invocare il metodo `next` ripetutamente, finché non sono stati restituiti tutti gli elementi, ma non c'è modo di "reimpostare" l'iteratore, facendolo tornare all'inizio della sequenza.

Una struttura dati che, però, voglia consentire ripetute scansioni può mettere a disposizione un metodo che restituisca un *nuovo* iteratore ogni volta che viene invocato. Per rendere standard questo comportamento, Java definisce un'altra interfaccia parametrica, `Iterable`, che contiene questo unico metodo:

`iterator()`: Restituisce un iteratore per gli elementi del contenitore.

Un esemplare di un tipico contenitore della libreria di Java, come `ArrayList`, è *iterabile* o scansionabile (che non vuol dire che sia esso stesso un *iteratore*): produce un iteratore per la propria raccolta di elementi come valore restituito dal metodo `iterator()`. Ogni invocazione di `iterator()` restituisce un nuovo esemplare di iteratore, consentendo così più scansioni (anche simultanee) del contenitore.

L'interfaccia `Iterable`, in Java, svolge anche un ruolo fondamentale come supporto del ciclo `for-each` (la cui sintassi e semantica sono state descritte nel Paragrafo 1.5.2). La sintassi:

```
for (tipoDiElemento nomeVariabile : contenitore) {
    corpoDelCiclo      // può fare riferimento a nomeVariabile
}
```

è valida per qualunque esemplare (*contenitore*) di una classe iterabile, cioè che implementi *Iterable*. Il *tipoDiElemento* deve essere il tipo degli oggetti restituiti dall'iteratore della classe (che è l'oggetto restituito dall'invocazione di *iterator()*), mentre *nomeVariabile* assumerà come valore, all'interno del *corpoDelCiclo*, gli elementi restituiti dall'iteratore, uno dopo l'altro a ogni iterazione del ciclo, in sequenza. In pratica, quella sintassi è un'abbreviazione per questa:

```
Iterator<tipoDiElemento> iter = contenitore.iterator();
while (iter.hasNext()) {
    tipoDiElemento nomeVariabile = iter.next();
    corpoDelCiclo      // può fare riferimento a nomeVariabile
}
```

Osserviamo che il metodo *remove* dell'iteratore non può essere invocato quando si usa la sintassi del ciclo *for-each*: bisogna, in tal caso, usare esplicitamente un iteratore. Come esempio, il ciclo seguente può essere utilizzato per eliminare tutti i numeri negativi da un esemplare di *ArrayList* contenente valori in virgola mobile.

```
ArrayList<Double> data; // da popolare con numeri casuali
Iterator<Double> walk = data.iterator();
while (walk.hasNext())
    if (walk.next() < 0.0)
        walk.remove();
```

7.4.2 Implementazione di iteratori

Per l'implementazione di iteratori si seguono, in generale, due stili diversi, in base a ciò che avviene nel momento in cui ne vengono creati esemplari e ogni volta che l'iteratore viene fatto avanzare da un'invocazione di *next()*.

Un *iteratore a fotografia* (*snapshot iterator*) gestisce una propria copia privata della sequenza di elementi, che viene costruita nel momento in cui l'oggetto iteratore viene creato. A tutti gli effetti l'iteratore "fa una fotografia" della sequenza di elementi presenti nel contenitore in quel momento e, quindi, eventuali modifiche che avvengano poi nel contenitore non influenzano il comportamento dell'iteratore. L'implementazione di iteratori che seguono questa strategia tende a essere semplice, perché è sufficiente fare una scansione della struttura principale nel momento in cui si costruisce un esemplare di iteratore. Lo svantaggio è che tale costruzione richiede un tempo $O(n)$, per copiare gli n elementi, e l'esemplare di iteratore occupa uno spazio $O(n)$, diverso da quello occupato dalla sequenza primaria, per memorizzare, appunto, gli elementi copiati.

Un *iteratore pigro* (*lazy iterator*) non fa una copia della sequenza, ma scandisce la struttura principale, passo dopo passo, soltanto quando richiesto dall'invocazione del metodo *next()*, per accedere all'elemento successivo. Il vantaggio di questa strategia di implementazione degli iteratori è che solitamente si può fare in modo che un esemplare di iteratore occupi uno spazio $O(1)$ e venga costruito in un tempo $O(1)$. Di converso, uno svantaggio dell'iteratore pigro (che a volte può, invece, essere una caratteristica voluta) è che il suo comportamento subisce gli effetti di eventuali modifiche apportate alla struttura dati su cui opera (da metodi che non siano il metodo *remove* dell'iteratore stesso) prima che la scansione sia terminata. Molti degli iteratori presenti nella libreria di Java

implementano un comportamento di tipo "fail fast" (letteralmente "fallisci in fretta") che rende immediatamente non utilizzabile un esemplare di iteratore qualora il contenitore su cui si basa subisca modifiche inaspettate.

A titolo di esempi, vedremo come implementare iteratori per le classi `ArrayList` e `LinkedList`. In entrambi i casi realizzeremo iteratori "pigri", dotati di supporto per l'operazione `remove` (ma senza la garanzia di "fallire in fretta").

Iteratore per esemplari di `ArrayList`

Iniziamo parlando della scansione di un esemplare della classe `ArrayList<E>`. Vogliamo che tale classe implementi l'interfaccia `Iterable<E>` (in effetti, tale requisito fa già parte dell'interfaccia `List` di Java), quindi dobbiamo aggiungere la definizione del metodo `iterator()`, che restituisca un esemplare di un oggetto che implementa l'interfaccia `Iterator<E>`. Con questo obiettivo in mente, definiamo una nuova classe, `ArrayIterator`, come classe non statica annidata all'interno di `ArrayList` (si tratta, quindi, di una *classe interna, inner class*, così come descritto nel Paragrafo 2.6). Il vantaggio di definire l'iteratore come classe interna è che in questo modo può avere accesso ai campi privati (come l'array `data`) che sono membri della classe "lista" che la contiene.

Il Codice 7.13 riporta la nostra implementazione. Il metodo `iterator()` di `ArrayList` restituisce un nuovo esemplare della classe interna `ArrayIterator`. Ogni esemplare di iteratore gestisce un proprio campo, `j`, che rappresenta l'indice del successivo elemento che verrà restituito: viene inizializzato a 0 e, quando assume il valore uguale alla dimensione della lista, non ci sono più elementi da restituire. Per consentire la rimozione di elementi tramite l'iteratore, utilizziamo anche una variabile booleana che segnala se è attualmente ammисibile un'invocazione di `remove`, oppure no.

Codice 7.13: Codice per la gestione di iteratori nella classe `ArrayList` (va inserito nella classe `ArrayList` definita nel Codice 7.2 e 7.3).

```

1 //-----+----- classe ArrayIterator annidata -----+
2 /**
3 * Una classe interna non static. Ogni esemplare contiene un riferimento
4 *隐式的 alla lista a cui appartiene, consentendo l'accesso ai suoi membri.
5 */
6 private class ArrayIterator implements Iterator<E> {
7     private int j = 0;           // indice del prossimo elemento da restituire
8     private boolean removable = false; // si può invocare remove ora?
9
10 /**
11 * Verifica se l'iteratore ha ancora oggetti da restituire.
12 * @return true se e solo se ci sono ancora oggetti da restituire
13 */
14 public boolean hasNext() { return j < size; } // size è un campo della lista
15
16 /**
17 * Restituisce l'oggetto successivo presente nell'iteratore.
18 *
19 */
20 * @return l'oggetto successivo
21 * @throws NoSuchElementException se non ci sono ulteriori elementi
22 */
23 public E next() throws NoSuchElementException {
24     if (j == size) throw new NoSuchElementException("No next element");
25     removable = true; // questo elemento potrà poi essere rimosso

```

```

25     return data[j++]; // post-incremento di j, così è pronto per il futuro
26 }
27
28 /**
29 * Elimina l'elemento restituito dalla più recente invocazione di next.
30 * @throws IllegalStateException se next non è mai stato invocato
31 * @throws IllegalStateException se remove è già stato invocato dopo next
32 */
33 public void remove() throws IllegalStateException {
34     if (!removable) throw new IllegalStateException("nothing to remove");
35     ArrayList.this.remove(j-1); // questo è l'ultimo che è stato restituito
36     j--;                      // il prossimo elemento da restituire è più a sinistra
37     removable = false;        // non si può invocare di nuovo remove prima di next
38 }
39 } //--- fine della classe ArrayIterator annidata -----
40
41 /**
42 * Restituisce un iteratore per gli elementi della lista. *
43 */
44 public Iterator<E> iterator() {
45     return new ArrayIterator(); // crea un nuovo esemplare della classe interna
46 }

```

Iteratore per esemplari di `LinkedPositionalList`

Per definire un iteratore per la classe `LinkedPositionalList` bisogna prima rispondere a una domanda: l'iteratore deve restituire gli *elementi* della lista o le sue *posizioni*? Se consentiamo all'utilizzatore della lista di fare una scansione delle sue posizioni, queste potrebbero essere utilizzate per accedere agli elementi contenuti al loro interno, per cui l'iterazione sulle posizioni è più generale. Tuttavia, l'approccio standard per una classe di tipo contenitore è quello di fornire supporto all'iterazione dei suoi elementi, in modo da poter usare il ciclo `for-each` per scrivere codice come questo:

```
for (String guest : waitlist)
```

nell'ipotesi che la variabile `waitlist` sia di tipo `LinkedPositionalList<String>`.

Per agevolare al massimo l'utilizzatore della lista, forniremo supporto per *entrambe* le forme di iterazione. Il metodo `iterator()`, come normalmente avviene, restituirà un iteratore che agisce sugli elementi della lista, per cui la nostra classe `LinkedPositionalList` implementerà in modo formale l'interfaccia `Iterable` del tipo dichiarato come tipo degli elementi della lista.

Per chi desidera scandire le posizioni della lista, metteremo inoltre a disposizione un nuovo metodo, `positions()`. A prima vista potrebbe sembrare naturale che questo metodo restituisca un esemplare di `Iterator`, ma preferiamo che restituisca un esemplare di un oggetto che sia `Iterable` (e, quindi, disponga di un proprio metodo `iterator()`, il quale, a sua volta, restituisca un iteratore di posizioni). Il motivo per questo ulteriore livello di complessità è il desiderio di consentire agli utilizzatori della nostra classe di poter scrivere un ciclo `for-each` con questa semplice sintassi:

```
for (Position<String> p : waitlist.positions())
```

Perché questa sintassi sia valida, il tipo restituito dal metodo `positions()` deve essere `Iterable`.

Il Codice 7.14 presenta quanto va aggiunto per consentire la scansione di elementi e posizioni di un oggetto di tipo `LinkedPositionalList`. Definiamo tre nuove classi interne. La prima di queste è `PositionIterator`, che fornisce il nucleo della funzionalità per scandire la

nostra lista: così come l'iteratore per liste con indice memorizzava come proprio campo l'indice del successivo elemento da restituire, questa classe gestisce la posizione dell'elemento successivo (oltre alla posizione dell'elemento restituito più di recente, per consentire il funzionamento del metodo di rimozione).

Per fare in modo che, come abbiamo deciso, il metodo `positions()` restituisca un oggetto di tipo `Iterable`, definiamo una classe interna banale, `PositionIterable`, che semplicemente costruisce e restituisce un nuovo oggetto di tipo `PositionIterator` ogni volta che viene invocato il suo metodo `iterator()`. Poi, il metodo `positions()` della classe esterna restituisce un nuovo esemplare di `PositionIterable`. Questa nostra infrastruttura realizzativa si basa fortemente sul fatto che queste classi siano classi interne (cioè classi annidate non statiche) e non semplici classi statiche annidate.

Infine, dobbiamo fare in modo che il metodo `iterator()` della classe esterna restituisca un iteratore di elementi (e non di posizioni). Invece di reinventare la ruota, come si suol dire, abbiamo banalmente adattato la classe `PositionIterator`, definendo una nuova classe, `ElementIterator`, che gestisce un esemplare di iteratore di posizioni, restituendo, uno dopo l'altro, gli elementi memorizzati in ciascuna posizione, ogni volta che viene invocato il suo metodo `next()`.

Codice 7.14: Codice per la gestione di iteratori di posizioni e di elementi nella classe `LinkedPositionalList` (va inserito nella classe `LinkedPositionalList` definita nel Codice 7.9, 7.10, 7.11 e 7.12).

```

1 //----- classe PositionIterator annidata -----
2 private class PositionIterator implements Iterator<Position<E>> {
3     private Position<E> cursor = first(); // prossima posizione da restituire
4     private Position<E> recent = null; // posizione restituita più recentemente
5     /** Verifica se l'iteratore ha altro da restituire. */
6     public boolean hasNext() { return (cursor != null); }
7     /** Restituisce la prossima posizione dell'iteratore. */
8     public Position<E> next() throws NoSuchElementException {
9         if (cursor == null) throw new NoSuchElementException("nothing left");
10        recent = cursor; // l'elemento in questa posizione potrebbe essere poi rimosso
11        cursor = after(cursor);
12        return recent;
13    }
14    /** Elimina l'elemento restituito dalla più recente invocazione di next. */
15    public void remove() throws IllegalStateException {
16        if (recent == null) throw new IllegalStateException("nothing to remove");
17        LinkedPositionalList.this.remove(recent); // elimina l'elemento dalla lista
18        recent = null;           // non si può invocare di nuovo remove prima di next
19    }
20 } //---- fine della classe PositionIterator annidata -----
21
22 //----- classe PositionIterable annidata -----
23 private class PositionIterable implements Iterable<Position<E>> {
24     public Iterator<Position<E>> iterator() { return new PositionIterator(); }
25 } //---- fine della classe PositionIterable annidata -----
26
27 /** Restituisce una rappresentazione iterabile delle posizioni della lista. */
28 public Iterable<Position<E>> positions() {
29     return new PositionIterable(); // crea un nuovo esemplare della classe interna
30 }
31

```

```

32 //----- classe ElementIterator annidata -----
33 /* Adatta l'iteratore restituito da positions() perché restituiscia elementi. */
34 private class ElementIterator implements Iterator<E> {
35     Iterator<Position<E>> posIterator = new PositionIterator();
36     public Boolean hasNext() { return posIterator.hasNext(); }
37     public E next() { return posIterator.next().getElement(); } // elemento!
38     public void remove() { posIterator.remove(); }
39 }
40
41 /** Restituisce un iteratore degli elementi memorizzati nella lista. */
42 public Iterator<E> iterator() { return new ElementIterator(); }

```

7.5 L'infrastruttura Java Collections Framework

La libreria di Java mette a disposizione molteplici strutture dati e interfacce, che compongono, tutte insieme, l'infrastruttura di elaborazione denominata *Java Collections Framework* (JCF), che fa parte del pacchetto `java.util` e contiene molte delle strutture dati trattate in questo libro: di alcune abbiamo già parlato, di altre discuteremo più avanti. L'interfaccia che funge da capostipite per l'intera infrastruttura prende il nome di `Collection`: si tratta di una generica interfaccia adatta a qualunque struttura dati che rappresenti una collezione (o raccolta) di elementi, come, ad esempio, una lista. L'interfaccia `Collection` contiene molti metodi, compresi alcuni che abbiamo già visto (come `size()`, `isEmpty()` e `iterator()`), ed è la super-interfaccia per altre interfacce del JCF che definiscono, nel pacchetto `java.util`, contenitori di elementi, come `Deque`, `List` e `Queue`, oltre ad altre che vedremo, come `Set` (nel Paragrafo 10.5.1) e `Map` (nel Paragrafo 10.1).

Il Java Collections Framework contiene anche classi concrete che implementano varie interfacce, dando luogo a un'ampia combinazione di proprietà e delle corrispondenti rappresentazioni, alcune delle quali sono riassunte nella Tabella 7.3: abbiamo indicato quali interfacce, tra `Queue`, `Deque` o `List`, sono implementate da ciascuna classe. Sono elencate, poi, alcune proprietà comportamentali delle classi: alcune rendono obbligatorio l'utilizzo di un limite prefissato di capacità, altre invece lo consentono.

Tabella 7.3: Alcune classi presenti nel Java Collections Framework.

Classe	Interfacce			Proprietà			Supporto	
	Queue	Deque	List	Capacità limitata	Thread-Safe	Bloccante	Array	Lista
<code>ArrayBlockingQueue</code>	✓			✓	✓	✓	✓	
<code>LinkedBlockingQueue</code>	✓			✓	✓	✓		✓
<code>ConcurrentLinkedQueue</code>	✓				✓		✓	
<code>ArrayDeque</code>	✓	✓					✓	
<code>LinkedBlockingDeque</code>	✓	✓		✓	✓	✓		✓
<code>ConcurrentLinkedDeque</code>	✓	✓			✓			✓
<code>ArrayList</code>			✓				✓	
<code>LinkedList</code>	✓	✓	✓					✓

Le classi più robuste forniscono supporto alla *concorrenza (concurrency)*, consentendo a più processi di condividere l'uso di una struttura dati in modo *thread-safe*, cioè "sicuro per thread" (un concetto che richiede alcune conoscenze di sistemi operativi). Se la struttura viene segnalata come *bloccante (blocking)*, un'invocazione che cerchi di accedere (per ispezione o rimozione) a un elemento di un contenitore vuoto rimane in attesa (*bloccando* il processo che ha eseguito l'invocazione) che un altro processo inserisca un elemento nel contenitore stesso. Analogamente, un'invocazione che cerchi di inserire un elemento in una struttura bloccante già piena rimarrà in attesa finché non si liberi lo spazio necessario.

7.5.1 Iteratori di lista in Java

La classe `java.util.LinkedList` non rende pubblico nella propria API un concetto di posizione che possa essere usato dai suoi utilizzatori, come invece abbiamo fatto nel nostro ADT "lista posizionale". Il modo preferito, in Java, per compiere ispezioni o modifiche a un oggetto di tipo `LinkedList`, senza usare indici, prevede l'utilizzo di un oggetto di tipo `ListIterator`, che viene restituito dal metodo `listIterator()`. Un tale iteratore mette a disposizione metodi per la scansione in avanti e all'indietro, oltre a metodi per effettuare aggiornamenti locali, in corrispondenza della posizione in cui si trova l'iteratore all'interno della lista. L'iteratore immagina che la propria posizione si trovi inizialmente prima del primo elemento della lista, poi tra due elementi e, infine, dopo l'ultimo elemento: usa, quindi, un *cursore* all'interno della lista, in modo molto simile al cursore visibile sullo schermo quando si utilizza un *editor* di testo, che viene posizionato e visualizzato tra due caratteri consecutivi. In particolare, l'interfaccia `java.util.ListIterator` dichiara i seguenti metodi:

- `add(e)`: Aggiunge l'elemento *e* in corrispondenza della posizione attuale.
- `hasNext()`: Restituisce `true` se e solo se c'è almeno un altro elemento dopo la posizione attuale.
- `hasPrevious()`: Restituisce `true` se e solo se c'è almeno un altro elemento prima della posizione attuale.
- `previous()`: Restituisce l'elemento *e* che si trova prima della posizione attuale e posiziona l'iteratore immediatamente prima di *e*.
- `next()`: Restituisce l'elemento *e* che si trova dopo la posizione attuale e posiziona l'iteratore immediatamente dopo *e*.
- `nextIndex()`: Restituisce l'indice dell'elemento che segue la posizione attuale.
- `previousIndex()`: Restituisce l'indice dell'elemento che precede la posizione attuale.
- `remove()`: Elimina l'elemento restituito dalla più recente invocazione di `next` o `previous`.
- `set(e)`: Sostituisce con *e* l'elemento restituito dalla più recente invocazione di `next` o `previous`.

Utilizzare più iteratori di questo tipo sulla stessa lista, modificandone il contenuto, è un'operazione rischiosa: se si devono effettuare inserimenti, rimozioni o sostituzioni in più punti di una lista, è più sicuro specificare tali punti usando delle posizioni. Purtroppo, però, la classe `java.util.LinkedList` non rende pubbliche le proprie posizioni ai suoi utilizzatori, quindi, per evitare i rischi derivanti dalle modifiche apportate a una lista che abbia creato diversi propri iteratori, questi hanno un comportamento di tipo *fail-fast*, che li rende non più operativi nel

momento in cui la lista su cui operano sia stata modificata da altri. Se, ad esempio, un oggetto L di tipo `java.util.LinkedList` ha restituito cinque diversi iteratori e uno di questi modifica L , nel momento in cui si tenti di utilizzare uno degli altri quattro iteratori verrà lanciata un'eccezione di tipo `ConcurrentModificationException`. In pratica, Java consente di avere più iteratori che operano una scansione su una medesima lista concatenata L , ma se uno di questi modifica L (usando uno dei metodi `add`, `set` o `remove` dell'iteratore stesso), allora tutti gli altri iteratori di L non saranno più operativi. Analogamente, se L viene modificata mediante uno dei propri metodi di aggiornamento, tutti i suoi iteratori non saranno più operativi.

7.5.2 Confronto con il nostro ADT “lista posizionale”

L'interfaccia `java.util.List` della libreria di Java mette a disposizione funzionalità simili a quelle dei nostri tipi di dati astratti “lista con indice” e “lista posizionale”; tale interfaccia è implementata mediante un array nella classe `java.util.ArrayList` e mediante una lista concatenata nella classe `java.util.LinkedList`.

Tabella 7.4: Corrispondenze tra i metodi del nostro ADT “lista posizionale” e le interfacce `List` e `ListIterator` del pacchetto `java.util`. Abbiamo usato A e L come abbreviazioni per `java.util.ArrayList` e `java.util.LinkedList` (o i loro tempi d'esecuzione).

Metodo dell'ADT lista posizionale	Metodo di <code>java.util.List</code>	Metodo di <code>ListIterator</code>	Commenti
<code>size()</code>	<code>size()</code>		Tempo $O(1)$
<code>isEmpty()</code>	<code>isEmpty()</code>		Tempo $O(1)$
		<code>get(i)</code>	A è $O(1)$, L è $O(\min\{i, n-i\})$
<code>first()</code>		<code>listIterator()</code>	Il primo elemento è il successivo
<code>last()</code>		<code>listIterator(size())</code>	L'ultimo elemento è il precedente
<code>before(p)</code>		<code>previous()</code>	Tempo $O(1)$
<code>after(p)</code>		<code>next()</code>	Tempo $O(1)$
<code>set(p, e)</code>		<code>set(e)</code>	Tempo $O(1)$
		<code>set(i, e)</code>	A è $O(1)$, L è $O(\min\{i, n-i\})$
		<code>add(i, e)</code>	Tempo $O(n)$
<code>addFirst(e)</code>		<code>add(0, e)</code>	A è $O(n)$, L è $O(1)$
<code>addFirst(e)</code>		<code>addFirst(e)</code>	Esiste solo in L , $O(1)$
<code>addLast(e)</code>		<code>add(e)</code>	Tempo $O(1)$
<code>addLast(e)</code>		<code>addLast(e)</code>	Esiste solo in L , $O(1)$
<code>addAfter(p, e)</code>		<code>add(e)</code>	Inserimento al cursore; A è $O(n)$, L è $O(1)$
<code>addBefore(p, e)</code>		<code>add(e)</code>	Inserimento al cursore; A è $O(n)$, L è $O(1)$
<code>remove(p)</code>		<code>remove()</code>	Rimozione al cursore; A è $O(n)$, L è $O(1)$
		<code>remove(i)</code>	A è $O(1)$, L è $O(\min\{i, n-i\})$

Inoltre, Java usa gli iteratori per ottenere una funzionalità simile a quella che la nostra lista posizionale ricava dalle posizioni. La Tabella 7.4 evidenzia le corrispondenze esistenti tra i metodi del nostro ADT “lista posizionale” e quelli delle interfacce `List` e `ListIterator`, con commenti relativi alle classi `ArrayList` e `LinkedList` del pacchetto `java.util`.

7.5.3 Algoritmi basati su liste nel Java Collections Framework

Nel Java Collections Framework, oltre alle classi, troviamo anche alcuni semplici algoritmi, implementati come metodi statici nella classe `java.util.Collections` (da non confondere con l’interfaccia `java.util.Collection`), tra i quali:

copy(L_{dest} , L_{src}): Copia tutti gli elementi della lista L_{src} (la lista sorgente, *source*) negli indici corrispondenti della lista L_{dest} (la lista destinazione, *destination*).

disjoint(C , D): Restituisce `true` se e solo se le collezioni C e D sono disgiunte, cioè non hanno alcun elemento uguale.

fill(L , e): Sostituisce con l’elemento e ciascun elemento della lista L , cioè “riempie (fill) L con e ”.

frequency(C , e): Restituisce il numero di elementi della collezione C che sono uguali a e .

max(C): Restituisce il massimo elemento presente nella collezione C , basandosi sull’ordinamento naturale tra i suoi elementi.

min(C): Restituisce il minimo elemento presente nella collezione C , basandosi sull’ordinamento naturale tra i suoi elementi.

replaceAll(L , e , f): Sostituisce con l’elemento f tutti gli elementi di L che sono uguali a e .

reverse(L): Inverte l’ordine degli elementi della lista L .

rotate(L , d): Ruota di d posti in modo circolare gli elementi della lista L (d può essere negativo).

shuffle(L): Permuta in modo pseudocasuale l’ordine degli elementi della lista L .

sort(L): Ordina la lista L , usando l’ordinamento naturale tra i suoi elementi.

swap(L , i , j): Scambia tra loro gli elementi di indice i e j nella lista L .

Convertire liste in array

Le liste corrispondono a un’interessante astrazione e si possono applicare a molti contesti diversi, ma ci sono situazioni in cui sarebbe utile poterle manipolare come se fossero array. Fortunatamente l’interfaccia `java.util.Collection` contiene i seguenti metodi, utili per generare un array che contenga gli stessi elementi della collezione data:

toArray(): Restituisce un array di elementi di tipo `Object` contenente tutti gli elementi di questa collezione.

toArray(A): Restituisce un array di elementi dello stesso tipo di A contenente tutti gli elementi di questa collezione.

Se la collezione è una lista, l'array restituito avrà gli elementi memorizzati nello stesso ordine della lista originaria. In questo modo, se si dispone di un metodo che elabora array e lo si vuole utilizzare con una lista o un altro tipo di contenitore, lo si può fare semplicemente usando il metodo `toArray()` di quel contenitore per generare una rappresentazione del suo contenuto sotto forma di array.

Convertire array in liste

In modo analogo, spesso è utile convertire un array nella lista equivalente, compito per il quale la classe `java.util.Arrays` contiene il metodo seguente:

`asList(A)`: Restituisce una lista avente lo stesso contenuto dell'array *A*, con elementi dello stesso tipo degli elementi di *A*.

La lista restituita da questo metodo usa l'array *A* come rappresentazione interna della lista stessa, per cui si tratta di una lista basata su array e ogni modifica apportata alla lista si rifletterà automaticamente nel contenuto di *A*. A causa di questo tipo di effetto collaterale, bisogna usare il metodo `asList` con molta attenzione, per evitare, appunto, conseguenze impreviste. Se usato con cura, però, questo metodo può spesso consentire un grande risparmio di codice. Ad esempio, il frammento seguente si può usare per mescolare in modo casuale l'array *arr* di oggetti di tipo `Integer`:

```
Integer[] arr = {1, 2, 3, 4, 5, 6, 7, 8}; // usa l'auto-boxing
List<Integer> listArr = Arrays.asList(arr);
Collections.shuffle(listArr); // effetto collaterale: mescola arr
```

È bene notare che l'array *A* passato come argomento al metodo `asList` deve essere un array di riferimenti (infatti nell'esempio precedente abbiamo usato un array di `Integer` e non di `int`), perché l'interfaccia `List` usa la programmazione generica e richiede che il tipo usato per gli elementi sia un oggetto.

7.6 Ordinare una lista posizionale

Nel Paragrafo 3.1.2 abbiamo presentato l'algoritmo di *ordinamento per inserimento* nel contesto di una sequenza basata su array, mentre in questo paragrafo svilupperemo un'implementazione di ordinamento operante su un esemplare di `PositionalList` e basata, ad alto livello, sullo stesso algoritmo, nel quale ciascun elemento viene via via posizionato in ordine rispetto a una collezione di dimensione crescente di elementi già ordinati.

Usiamo la variabile `marker` (*marcatore*) per rappresentare la posizione più a destra della porzione di lista ordinata. Ad ogni passo, consideriamo la posizione subito a destra del marcatore come *pivot* (cioè come "posizione in esame") e cerchiamo all'interno della porzione ordinata la posizione corretta per l'elemento che si trova in tale posizione *pivot*; infine, usiamo un'altra variabile, `walk`, per spostarci a sinistra a partire da `marker` finché c'è un elemento precedente il cui valore è maggiore di quello del *pivot*. La Figura 7.9 riporta una configurazione tipica di queste variabili e il Codice 7.15 presenta un'implementazione di questa strategia in Java.

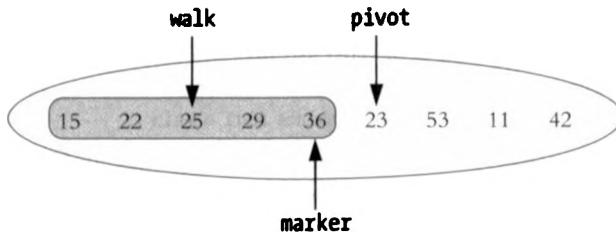


Figura 7.9: Rappresentazione di un passo del nostro algoritmo di ordinamento per inserimento. Gli elementi più a sinistra, fino a quello indicato da **marker** compreso, sono già stati ordinati. In questo passo, l'elemento indicato da **pivot** andrà posto immediatamente a sinistra della posizione indicata da **walk**.

Codice 7.15: Codice Java per eseguire l'ordinamento per inserimento di una lista posizionale.

```

1  /** Insertion-sort di una lista posizionale di interi in senso non decrescente */
2  public static void insertionSort(PositionalList<Integer> list) {
3      Position<Integer> marker = list.first(); // estremo destro della parte ordinata
4      while (marker != list.last()) {
5          Position<Integer> pivot = list.after(marker);
6          int value = pivot.getElement(); // numero da posizionare
7          if (value > marker.getElement()) // il pivot è già ordinato
8              marker = pivot;
9          else { // il pivot va spostato: cerca l'elemento...
10             Position<Integer> walk = marker; // più a sinistra che sia maggiore di value
11             while (walk != list.first() && list.before(walk).getElement() > value)
12                 walk = list.before(walk);
13             list.remove(pivot); // elimina il pivot e
14             list.addBefore(walk, value); // lo reinserisce subito prima di walk
15         }
16     }
17 }
```

7.7 Caso di studio: gestire frequenze di accesso

Il tipo di dato astratto “lista posizionale” è utile in numerosi contesti. Ad esempio, un programma che simula un gioco di carte potrebbe rappresentare le carte in mano a un giocatore usando una lista posizionale (Esercizio P-7.60). Dato che molte persone tengono in mano le carte in modo che quelle dello stesso seme stiano vicine, l’inserimento e la rimozione di carte dalla mano di un giocatore si potrebbe implementare usando i metodi della lista posizionale, con le posizioni individuate da un ordinamento tra i semi. Analogamente, un semplice editor di testi usa in modo naturale l’inserimento e la rimozione posizionali, perché tipicamente esegue tutte le modifiche relativamente a un *cursore*, che rappresenta la posizione attuale all’interno della lista di caratteri che costituisce il testo in fase di elaborazione.

In questo paragrafo vedremo come si possa gestire una raccolta di elementi tenendo traccia del numero di accessi compiuti nei confronti di ciascun singolo elemento. La memorizzazione di questi conteggi di accessi ci consente di sapere quali elementi siano più popolari: tra gli esempi di questo scenario, possiamo immaginare un browser web che tenga traccia delle pagine più visitate da un utente, oppure una raccolta di brani musicali che

gestisce una lista delle canzoni più ascoltate da parte di un utente. Useremo come modello di queste situazioni un nuovo tipo di dato astratto, la *lista dei preferiti* (*favorites list*), che fornisce supporto per i metodi `size` e `isEmpty`, oltre ai seguenti:

- access(*e*):** Accede all'elemento *e*, aggiungendolo al contenitore se non è già presente, per poi incrementare il relativo conteggio.
- remove(*e*):** Elimina l'elemento *e* dal contenitore, se è presente.
- getFavorites(*k*):** Restituisce un contenitore iterabile con i *k* elementi che hanno avuto più accessi.

7.7.1 Implementazione con una lista ordinata

Il nostro primo approccio per la gestione di una lista dei preferiti memorizza gli elementi in una lista concatenata, conservandoli in ordine non crescente di numero di accessi. Eseguiamo l'accesso a un elemento o la sua rimozione scandendo la lista a partire dall'elemento che ha registrato il maggior numero di accessi, procedendo verso quello che ne ha registrati di meno. In questo modo, restituire la lista dei *k* elementi che hanno avuto più accessi è veramente facile, perché sono i primi *k* elementi della lista.

Per preservare la condizione invariante che prevede la memorizzazione degli elementi in ordine non crescente di numero di accessi, dobbiamo esaminare gli effetti che può avere su tale ordinamento una singola operazione di accesso. Il conteggio relativo all'elemento a cui si accede aumenta di un'unità, per cui può diventare maggiore di uno o più degli elementi che lo precedono nella lista, violando così l'invariante.

Fortunatamente, possiamo ristabilire la validità dell'invariante di ordinamento usando una tecnica simile a un singolo passo dell'algoritmo di ordinamento per inserimento, visto nel paragrafo precedente. Basta effettuare una scansione della lista all'indietro, a partire dalla posizione dell'elemento il cui conteggio è aumentato, fino a individuare una posizione valida in cui riposizionare l'elemento.

Lo schema progettuale di composizione

Vogliamo implementare una lista di preferiti usando, per la memorizzazione degli elementi, un oggetto di tipo `PositionalList`. Se gli elementi della lista posizionale fossero semplicemente gli elementi della lista dei preferiti, sarebbe veramente difficile gestire i conteggi degli accessi, preservandone l'associazione corretta con gli elementi quando il contenuto della lista viene riordinato. Per questo motivo usiamo, invece, uno schema generale di progettazione orientata agli oggetti, lo *schema di composizione* (*composition pattern*), nel quale definiamo un singolo oggetto composto da due o più altri oggetti (si veda, per esempio, il Paragrafo 2.5.2).

In particolare, definiamo una classe annidata non pubblica, `Item` (un sinonimo di “elemento” generico), i cui esemplari memorizzano un elemento della lista dei preferiti e il relativo conteggio di accessi. Poi, gestiamo la lista dei preferiti come un oggetto di tipo `PositionalList` di esemplari di `Item`, in modo che, in questa nostra rappresentazione, il conteggio relativo a un elemento sia memorizzato assieme all'elemento stesso (il tipo di dato `Item` non è mai reso disponibile all'utilizzatore di un esemplare della nostra `FavoritesList`).

Codice 7.16: La classe FavoritesList (prosegue nel Codice 7.17).

```

1  /** Gestisce una lista di elementi ordinati in base alla frequenza di accesso. */
2  public class FavoritesList<E> {
3      //----- classe Item annidata -----
4      protected static class Item<E> {
5          private E value;
6          private int count = 0;
7          /** Costruisce un nuovo elemento con conteggio iniziale zero. */
8          public Item(E val) { value = val; }
9          public int getCount() { return count; }
10         public E getValue() { return value; }
11         public void increment() { count++; }
12     //----- fine della classe Item annidata -----
13
14     PositionalList<Item<E>> list = new LinkedPositionalList<>(); // lista di oggetti
15     public FavoritesList() {} // costruisce una lista di preferiti vuota
16
17     // metodi ausiliari non pubblici
18     /** Abbreviazione per accedere al preferito memorizzato in posizione p. */
19     protected E value(Position<Item<E>> p) { return p.getElement().getValue(); }
20
21     /** Abbreviazione per accedere al conteggio associato alla posizione p. */
22     protected int count(Position<Item<E>> p) { return p.getElement().getCount(); }
23
24     /** Restituisce la Position che ha value uguale a e (o null se non trovato). */
25     protected Position<Item<E>> findPosition(E e) {
26         Position<Item<E>> walk = list.first();
27         while (walk != null && !e.equals(value(walk)))
28             walk = list.after(walk);
29         return walk;
30     }

```

Codice 7.17: La classe FavoritesList (prosegue dal Codice 7.16).

```

31     /** Sposta verso sinistra l'oggetto in posizione p sulla base del conteggio. */
32     protected void moveUp(Position<Item<E>> p) {
33         int cnt = count(p); // conteggio aggiornato dell'elemento a cui si è acceduto
34         Position<Item<E>> walk = p;
35         while (walk != list.first() && count(list.before(walk)) < cnt)
36             walk = list.before(walk); // trovato un conteggio minore a sinistra
37         if (walk != p)
38             list.addBefore(walk, list.remove(p)); // elimina e reinserisce l'oggetto
39     }
40
41     // metodi pubblici
42     /** Restituisce il numero di elementi presenti nella lista dei preferiti. */
43     public int size() { return list.size(); }
44
45     /** Restituisce true se e solo se la lista dei preferiti è vuota. */
46     public boolean isEmpty() { return list.isEmpty(); }
47
48     /** Accede all'elemento e (eventualmente nuovo), incrementando il suo conteggio. */
49     public void access(E e) {
50         Position<Item<E>> p = findPosition(e); // cerca un elemento esistente
51         if (p == null)
52             p = list.addLast(new Item<E>(e)); // se nuovo, aggiunge alla fine

```

```

        p.getElement().increment();           // in ogni caso, incrementa il conteggio
        moveUp(p);                         // sposta a sinistra, se serve
    }

    /** Elimina dalla lista dei preferiti l'elemento uguale a e (se c'è). */
    public void remove(E e) {
        Position<Item<E>> p = findPosition(e); // cerca un elemento esistente
        if (p != null)
            list.remove(p);
    }

    /** Restituisce un contenitore iterabile con i k elementi aventi più accessi. */
    public Iterable<E> getFavorites(int k) throws IllegalArgumentException {
        if (k < 0 || k > size())
            throw new IllegalArgumentException("Invalid k");
        PositionalList<E> result = new LinkedPositionalList<E>();
        Iterator<Item<E>> iter = list.iterator();
        for (int j=0; j < k; j++)
            result.addLast(iter.next().getValue());
        return result;
    }
}

```

7.7.2 Uso di una lista con l'euristica *move-to-front*

La precedente implementazione di una lista di preferiti esegue il metodo `access(e)` in un tempo proporzionale all'indice di e all'interno della lista, cioè, se e è il k -esimo elemento più popolare nella lista dei preferiti, allora il suo accesso richiede un tempo $O(k)$. In molte sequenze di accessi effettuati nella vita reale (ad esempio, le pagine web visitate da un utente), dopo l'accesso a un determinato elemento aumenta la probabilità che si acceda di nuovo al medesimo elemento nell'immediato futuro. Si dice che situazioni di questo tipo sono caratterizzate dal fatto che i *riferimenti sono locali* (*locality of reference*).

La strategia *move-to-front* ("sposta all'inizio") è una strategia *euristica* (cioè una regola empirica) che cerca di trarre vantaggio dal fatto che, in una sequenza di accessi, i riferimenti siano locali. Per applicarla, ogni volta che viene effettuato l'accesso a un elemento, lo spostiamo all'inizio della lista, nella speranza, ovviamente, che nell'immediato futuro ci sia un nuovo accesso allo stesso elemento. Consideriamo, ad esempio, uno scenario in cui ci siano n elementi e la seguente sequenza di n^2 accessi:

- n accessi all'elemento 1
- n accessi all'elemento 2
- ...
- n accessi all'elemento n

Se memorizziamo gli elementi ordinandoli in base al numero di accessi, inserendo nella lista ciascun nuovo elemento nel momento in cui vi si effettua il primo accesso, otteniamo che:

- ogni accesso all'elemento 1 viene eseguito in un tempo $O(1)$
- ogni accesso all'elemento 2 viene eseguito in un tempo $O(2)$
- ...
- ogni accesso all'elemento n viene eseguito in un tempo $O(n)$

Quindi, il tempo totale speso per eseguire la sequenza di accessi è proporzionale a

$$n + 2n + 3n + \dots + n \cdot n = n(1 + 2 + 3 + \dots + n) = n \cdot \frac{n(n+1)}{2}$$

che è $O(n^3)$.

Se, invece, usiamo l'euristica *move-to-front*, inserendo nella lista ciascun nuovo elemento nel momento in cui vi si effettua il primo accesso, otteniamo che:

- ogni accesso all'elemento 1 (dopo l'inserimento) viene eseguito in un tempo $O(1)$
- ogni accesso all'elemento 2 (dopo l'inserimento) viene eseguito in un tempo $O(1)$
- ...
- ogni accesso all'elemento n (dopo l'inserimento) viene eseguito in un tempo $O(1)$

Quindi, il tempo totale speso per eseguire la sequenza di accessi è, in questo caso, $O(n^2)$. Possiamo così concludere che, in questa situazione, l'implementazione che usa *move-to-front* ha un tempo d'accesso complessivo minore. Tuttavia, l'approccio *move-to-front* è una strategia euristica, perché esistono sequenze di accessi che rendono tale approccio più lento rispetto alla semplice gestione della lista dei preferiti che conserva gli elementi ordinati in base al conteggio degli accessi.

I compromessi relativi all'euristica *move-to-front*

Se non conserviamo più gli elementi della lista dei preferiti ordinati in base al conteggio dei rispettivi accessi, quando ci viene chiesto di trovare i k elementi che hanno avuto il maggior numero di accessi dobbiamo cercarli. Implementeremo il metodo `getFavorites(k)` in questo modo:

1. Copiamo tutti gli elementi della lista dei preferiti in un'altra lista, che chiamiamo `temp`.
2. Effettuiamo k scansioni della lista `temp`. Durante ciascuna scansione, troviamo l'elemento avente il massimo valore del conteggio, lo eliminiamo da `temp` e lo aggiungiamo al risultato.

Questa implementazione del metodo `getFavorites(k)` richiede un tempo $O(kn)$, quindi, quando k è una costante, il metodo `getFavorites(k)` viene eseguito in un tempo $O(n)$: questo succede, ad esempio, quando vogliamo ottenere la "top ten", cioè l'elenco dei dieci elementi migliori. Se, invece, k è proporzionale a n , allora il metodo `getFavorites(k)` viene eseguito in un tempo $O(n^2)$: ad esempio, quando vogliamo la "top 25%", cioè il primo quartile della lista dei preferiti.

Nel Capitolo 9 presenteremo una struttura dati che ci consentirà di implementare il metodo `getFavorites(k)` in un tempo $O(n + k \log n)$, come descritto nell'Esercizio P-9.51; con tecniche ancora più avanzate si può ottenere un tempo $O(n + k \log k)$.

Potremmo facilmente ottenere un tempo $O(n \log n)$ usando un algoritmo di ordinamento standard per riordinare la lista temporanea prima di restituire i k elementi aventi conteggio maggiore (come descritto nel Capitolo 12): questo approccio sarebbe preferibile nel caso in cui k fosse $\Omega(\log n)$, ricordando la notazione introdotta nel Paragrafo 4.3.1 per

descrivere un limite inferiore asintotico al tempo d'esecuzione di un algoritmo. Infine, esiste un algoritmo di ordinamento specializzato (si veda il Paragrafo 12.3.2) che può trarre vantaggio dal fatto che i conteggi degli accessi sono numeri interi, riuscendo a eseguire il metodo `getFavorites` in un tempo $O(n)$ per qualunque valore di k .

Implementare l'euristica *move-to-front* in Java

Nel Codice 7.18 forniamo un'implementazione in Java di una lista di preferiti usando l'e-
ristica *move-to-front*. La nuova classe `FavoritesListMTF` eredita la maggior parte della propria
funzionalità dalla classe originaria, `FavoritesList`, che funge da superclasse.

Nel nostro primo progetto, il metodo `access` della lista di preferiti si basa sul metodo ausiliario `moveUp`, avente accesso `protected`, che ha il compito di far scorrere un elemento verso l'inizio della lista, se questo si rende necessario in conseguenza dell'incremento del suo conteggio di accessi. Quindi, per implementare l'euristica `move-to-front` ci basta semplicemente sovrascrivere quel metodo in modo che dopo ogni accesso a un elemento, questo venga portato direttamente all'inizio della lista (se non vi si trova già), un'azione veramente facile da implementare in una lista posizionale.

La parte più complessa della nostra classe `FavoritesListMTF` è la nuova definizione del metodo `getFavorites`. Ci basiamo sul primo degli approcci delineati in precedenza, inserendo copie degli oggetti in una lista temporanea, per poi ripetutamente trovare, per k volte, l'elemento che ha il conteggio maggiore tra quelli rimasti nella lista temporanea e rimuoverlo, inserendolo nella lista che verrà restituita.

Codice 7.18: La classe `FavoritesListMTF` che implementa l'euroistica *move-to-front*. Questa classe estende `FavoritesList` (descritta nel Codice 7.16 e 7.17), sovrascrivendone i metodi `moveUp` e `getFavorites`.

```
1  /** Gestisce una lista di preferiti usando l'euristica move-to-front. */
2  public class FavoritesListMTF<E> extends FavoritesList<E> {
3
4      /** Sposta all'inizio della lista l'oggetto in posizione p a cui si è acceduto. */
5      protected void moveUp(Position<Item<E>> p) {
6          if (p != list.first())
7              list.addFirst(list.remove(p)); // elimina e reinserisce all'inizio
8      }
9
10     /** Restituisce un contenitore iterabile con i k elementi aventi più accessi. */
11     public Iterable<E> getFavorites(int k) throws IllegalArgumentException {
12         if (k < 0 || k > size())
13             throw new IllegalArgumentException("Invalid k");
14
15         // iniziamo facendo una copia della lista originale
16         PositionalList<Item<E>> temp = new LinkedPositionalList<>();
17         for (Item<E> item : list)
18             temp.addLast(item);
19
20         // cerchiamo ed eliminiamo ripetutamente l'elemento con il conteggio massimo
21         PositionalList<E> result = new LinkedPositionalList<>();
22         for (int j=0; j < k; j++) {
23             Position<Item<E>> highPos = temp.first();
24             Position<Item<E>> walk = temp.after(highPos);
25             while (walk != null) {
26                 if (count(walk) > count(highPos))
27                     highPos = walk;
28                 walk = temp.after(walk);
29             }
30             temp.remove(highPos);
31         }
32     }
33
34     private int count(Position<Item<E>> pos) {
35         int count = 0;
36         for (Position<Item<E>> walk = pos; walk != null; walk = temp.after(walk))
37             count++;
38         return count;
39     }
40 }
```

```
        highPos = walk;
        walk = temp.after(walk);
    }
    // abbiamo trovato l'elemento con il conteggio massimo
    result.addLast(value(highPos));
    temp.remove(highPos);
}
return result;
}
```

7.8 Esercizi

Riepilogo e approfondimento

- R-7.1 Raffigurare, in modo simile a quanto fatto nell'Esempio 7.1, una lista L , inizialmente vuota, dopo l'esecuzione della sequenza di operazioni seguente: $\text{add}(0, 4)$, $\text{add}(0, 3)$, $\text{add}(0, 2)$, $\text{add}(2, 1)$, $\text{add}(1, 5)$, $\text{add}(1, 6)$, $\text{add}(3, 7)$, $\text{add}(0, 8)$.
- R-7.2 Definire un'implementazione dell'ADT "pila" usando una lista con indice come spazio di memorizzazione.
- R-7.3 Definire un'implementazione dell'ADT "coda doppia" usando una lista con indice come spazio di memorizzazione.
- R-7.4 Dimostrare la validità dei tempi d'esecuzione mostrati nella Tabella 7.1 per i metodi di una lista con indice implementata mediante un array (che non aumenta di dimensione).
- R-7.5 La classe `java.util.ArrayList` contiene un metodo, `trimToSize()`, che sostituisce l'array usato internamente come spazio di memorizzazione con un altro la cui capacità sia esattamente uguale al numero di elementi presenti nella lista. Implementare tale metodo per la nostra versione di `ArrayList` che usa un array dinamico, vista nel Paragrafo 7.2.
- R-7.6 Ripetere la dimostrazione della Proposizione 7.2 ipotizzando che per fare aumentare la dimensione dell'array da k a $2k$ servano $3k$ ciber-dollari. Quanto si deve addebitare per ciascuna operazione *push* per far funzionare l'analisi ammortizzata?
- R-7.7 Prendere in esame un'implementazione dell'ADT "lista con indice" che usa un array dinamico, ma, nel momento in cui viene raggiunto il limite di capacità, invece di copiare gli elementi in un array di dimensione doppia (passando, cioè, da N a $2N$), si copiano gli elementi in un array avente $\lceil N/4 \rceil$ celle aggiuntive, aumentando la dimensione da N a $N + \lceil N/4 \rceil$. Dimostrare che anche in questo caso l'esecuzione di n operazioni *push* (cioè operazioni di inserimento in fondo alla lista) richiede un tempo $O(n)$.
- R-7.8 Ipotizzare che un contenitore C di elementi sia gestito in modo che, ogni volta che viene aggiunto un nuovo elemento, il contenuto di C viene copiato in una nuova lista con indice avente proprio la dimensione che serve. Qual è il tempo necessario per aggiungere n elementi a un contenitore di questo tipo, inizialmente vuoto?
- R-7.9 Il metodo `add` di una lista con indice che usa un array dinamico, visto nel Codice 7.5, è affetto da un'inefficienza: nel caso in cui avvenga un ridimensionamento, l'operazione richiede tempo per copiare tutti gli elementi dal vecchio array al

nuovo, dopodiché un ciclo, nel corpo del metodo `add`, fa scorrere alcuni elementi per far posto all'elemento nuovo. Fornire una migliore implementazione di tale metodo `add`, che, nel caso in cui avvenga un ridimensionamento, copi gli elementi direttamente nella loro posizione definitiva nel nuovo array (cioè senza che si renda necessario un loro scorrimento).

- R-7.10 Fornire una nuova implementazione della classe `ArrayStack`, vista nel Paragrafo 6.1.2, usando array dinamici per consentire una capacità non limitata.
- R-7.11 Descrivere un'implementazione dei metodi `addLast` e `addBefore` della lista posizionale realizzati usando soltanto i metodi appartenenti a questo insieme: `{isEmpty, first, last, before, after, addAfter, addFirst}`.
- R-7.12 Nell'ipotesi che si voglia estendere il tipo di dato astratto `PositionalList` con un metodo, `indexOf(p)`, che restituisca l'indice corrispondente all'elemento memorizzato nella posizione `p`, dimostrare come si possa implementare tale metodo usando soltanto altri metodi dell'interfaccia `PositionalList` (e non dettagli relativi alla nostra implementazione, la classe `LinkedPositionalList`).
- R-7.13 Nell'ipotesi che si voglia estendere il tipo di dato astratto `PositionalList` con un metodo, `findPosition(e)`, che restituisca la prima posizione contenente un elemento uguale a `e` (o `null` se non esiste una tale posizione), dimostrare come si possa implementare tale metodo usando soltanto altri metodi dell'interfaccia `PositionalList` (e non dettagli relativi alla nostra implementazione, la classe `LinkedPositionalList`).
- R-7.14 L'implementazione della classe `LinkedPositionalList` vista nel Codice 7.9, 7.10, 7.11 e 7.12, non cerca in alcun modo di controllare se una data posizione `p`, ricevuta come parametro di un metodo, sia effettivamente appartenente alla lista. Spiegare dettagliatamente gli effetti sulla lista `L` dell'invocazione `L.addAfter(p, e)`, nel caso in cui la posizione `p` appartenga a un'altra lista, `M`.
- R-7.15 Per rappresentare in modo migliore una coda FIFO in cui gli elementi possano essere eliminati prima che raggiungano l'estremità iniziale, progettare una classe, `LinkedPositionalQueue`, che fornisca completo supporto all'ADT coda, anche se con il metodo `enqueue` che restituisce un esemplare di posizione, e che sia dotata di un metodo aggiuntivo, `remove(p)`, che elimini dalla coda l'elemento associato alla posizione `p`. Si può usare lo schema progettuale di adattatore (visto nel Paragrafo 6.1.3), usando un esemplare di `LinkedPositionalList` come spazio interno di memorizzazione.
- R-7.16 Descrivere come si possa implementare in una lista posizionale un metodo, `alternateIterator()`, che fornisca un iteratore che, a sua volta, restituisca, uno dopo l'altro, soltanto gli elementi che, nella lista, hanno un indice pari.
- R-7.17 Riprogettare la classe `Progression`, vista nel Paragrafo 2.2.3, in modo che implementi formalmente l'interfaccia `Iterator<Long>`.
- R-7.18 L'interfaccia `java.util.Collection` contiene un metodo, `contains(o)`, che restituisce `true` se e solo se il contenitore contiene un oggetto uguale all'esemplare `o` di `Object`. Implementare tale metodo nella classe `ArrayList` del Paragrafo 7.2.
- R-7.19 L'interfaccia `java.util.Collection` contiene un metodo, `clear()`, che elimina tutti gli elementi dal contenitore. Implementare tale metodo nella classe `ArrayList` del Paragrafo 7.2.
- R-7.20 Spiegare come si possa utilizzare il metodo `java.util.Collections.reverse` per invertire il contenuto di un array di oggetti.

- R-7.21 Dato l'insieme di elementi $\{a, b, c, d, e, f\}$ memorizzati in una lista, descrivere lo stato finale della lista nell'ipotesi che usi l'euristica *move-to-front* e gli accessi agli elementi avvengano secondo la sequenza $(a, b, c, d, e, f, a, c, f, b, d, e)$.
- R-7.22 Nell'ipotesi di aver effettuato kn accessi agli elementi di una lista L contenente n elementi (con $k \geq 1$ e intero), qual è il numero minimo e il numero massimo di elementi che sono stati oggetto di un numero di accessi minore di k ?
- R-7.23 Data una lista L contenente n oggetti, gestita con l'euristica *move-to-front*, descrivere una sequenza di $O(n)$ accessi che inverta il contenuto di L .
- R-7.24 Implementare nella classe `FavoritesList` il metodo `resetCounts()` che azzeri i conteggi di accessi relativi a tutti gli elementi (lasciando inalterato l'ordine degli elementi nella lista).

Creatività

- C-7.25 Fornire un'implementazione di lista basata su array, con capacità fissa, che gestisca l'array in modalità circolare, in modo che impieghi un tempo $O(1)$ per eseguire inserimenti e rimozioni in corrispondenza dell'indice 0 e della fine della lista. L'implementazione deve anche avere un metodo `get` che sia tempo-costante.
- C-7.26 Completare l'esercizio precedente usando un array dinamico perché abbia una capacità non limitata.
- C-7.27 Modificare la nostra implementazione di `ArrayList` in modo che supporti l'interfaccia `Cloneable` descritta nel Paragrafo 3.6.
- C-7.28 Nel Paragrafo 7.5.3 abbiamo fatto vedere come si possa usare il metodo `Collections.shuffle` per mescolare un array di riferimenti. Fornire un'implementazione diretta di un metodo `shuffle` per un array di valori di tipo `int`. È consentito l'uso del metodo `nextInt(n)` della classe `Random`, che restituisce un numero intero casuale compreso tra 0 e $n - 1$, estremi compresi. Il metodo deve garantire che ogni possibile ordinamento ottenibile abbia la stessa probabilità. Qual è il tempo d'esecuzione del metodo?
- C-7.29 Modificare l'implementazione di lista con indice presentata nel Paragrafo 7.2.1 in modo che l'array dimezzi la propria dimensione quando il numero, n , di elementi presenti nell'array scende al di sotto di $N/4$, dove N è la capacità dell'array.
- C-7.30 Dimostrare che, quando si usa un array dinamico che aumenta e diminuisce la propria dimensione come descritto nell'esercizio precedente, la sequenza seguente di $2n$ operazioni richiede un tempo $O(n)$: n inserimenti alla fine di una lista inizialmente vuota, seguite da n rimozioni, tutte effettuate alla fine della lista.
- C-7.31 Dimostrare che qualunque sequenza di n operazioni `push` o `pop` (cioè inserimenti e rimozioni alla fine dell'array) eseguita su un array dinamico inizialmente vuoto che usi la strategia descritta nell'Esercizio C-7.29 richiede un tempo $O(n)$.
- C-7.32 Considerare una variante dell'Esercizio C-7.29 in cui un array di capacità N viene ridimensionato in modo che la sua capacità sia esattamente uguale al numero di elementi ogni volta che tale numero di elementi scende al di sotto di $N/4$. Dimostrare che qualunque sequenza di n operazioni `push` o `pop` eseguita su un array dinamico inizialmente vuoto richiede un tempo $O(n)$.
- C-7.33 Considerare una variante dell'Esercizio C-7.29 in cui un array di capacità N viene ridimensionato in modo che la sua capacità sia esattamente uguale al numero di elementi ogni volta che tale numero di elementi scende al di sotto di $N/2$.

Dimostrare che esiste una sequenza di n operazioni *push* e *pop* che richiede un tempo $\Omega(n^2)$.

- C-7.34 Descrivere come si possa implementare l'ADT coda usando due pile come variabili di esemplare in modo che tutte le operazioni della coda vengano eseguite in un tempo ammortizzato $O(1)$. Dimostrare il limite ammortizzato.
- C-7.35 Modificare la classe `ArrayList`, vista nel Paragrafo 6.2.2, usando array dinamici per consentire una capacità non limitata. Porre particolare attenzione alla gestione circolare dell'array nel momento in cui lo si ridimensiona.
- C-7.36 Nell'ipotesi di voler estendere l'interfaccia `PositionalList` in modo che contenga un metodo, `positionAtIndex(i)`, che restituisce la posizione dell'elemento avente indice i (oppure lancia un'eccezione di tipo `IndexOutOfBoundsException`, quando necessario), spiegare come si possa implementare tale metodo usando soltanto i metodi dell'interfaccia `PositionalList`, effettuando il numero appropriato di passi di una scansione della lista a partire dal suo inizio.
- C-7.37 Risolvere nuovamente il problema precedente, usando però la dimensione della lista per effettuare la scansione a partire dalla sua estremità più vicina all'indice in esame.
- C-7.38 Spiegare come qualunque implementazione dell'ADT `PositionalList` possa fornire supporto a tutti i metodi dell'ADT `List`, descritto nel Paragrafo 7.1, nell'ipotesi che all'implementazione venga aggiunto il metodo `positionAtIndex(i)` descritto nell'Esercizio C-7.36.
- C-7.39 Ipotizzare di voler estendere il tipo di dato astratto `PositionalList` con un metodo, `moveToFront(p)`, che sposta all'inizio della lista l'elemento che si trova nella posizione p (se p non è la posizione iniziale), preservando l'ordine relativo tra gli altri elementi della lista. Spiegare come si possa implementare tale metodo usando soltanto i metodi esistenti nell'interfaccia `PositionalList` (senza sfruttare i dettagli realizzativi della nostra classe `LinkedPositionalList`).
- C-7.40 Risolvere nuovamente il problema precedente modificando l'implementazione della classe `LinkedPositionalList` in modo che non crei o distrugga nessun nodo.
- C-7.41 Modificare la nostra implementazione della classe `LinkedPositionalList` in modo che supporti l'interfaccia `Cloneable`, descritta nel Paragrafo 3.6.
- C-7.42 Descrivere un metodo non ricorsivo per invertire il contenuto di una lista posizionale implementata mediante una lista doppiamente concatenata, usando un'unica scansione della lista stessa.
- C-7.43 Il Paragrafo 7.3.3 descrive una rappresentazione *basata su array* per implementare l'ADT lista posizionale. Dare una descrizione, mediante pseudocodice, del metodo `addBefore` adatto a tale rappresentazione.
- C-7.44 Descrivere un metodo per eseguire un *mescolamento delle carte* (*card shuffle*) su una lista contenente $2n$ elementi, convertendola in due liste. Un "mescolamento delle carte" è una particolare permutazione nella quale una lista L viene suddivisa in due liste, L_1 e L_2 , dove L_1 è la prima metà di L e L_2 è la seconda metà di L ; poi, queste due liste vengono fuse in una sola, prendendo il primo elemento di L_1 , seguito dal primo elemento di L_2 , seguiti a loro volta dal secondo elemento di L_1 , e, poi, dal secondo elemento di L_2 , e così via.
- C-7.45 Spiegare come si potrebbe modificare la classe `LinkedPositionalList` in modo che possa rilevare l'errore descritto nell'Esercizio R-7.14.

- C-7.46 Modificare la classe `LinkedPositionalList` in modo che metta a disposizione un metodo `swap(p, q)` che scambi tra loro i nodi corrispondenti alle posizioni p e q , ricollegando i nodi esistenti, senza creare nuovi nodi.
- C-7.47 Un array si dice *spars* se la maggior parte dei suoi elementi è `null`. Per implementare in modo efficiente un tale array, A , si può usare una lista, L . Nello specifico, per ogni cella $A[i]$ che non contiene il riferimento `null`, possiamo memorizzare in L una coppia, (i, e) , dove e è l'elemento memorizzato in $A[i]$. Questo approccio consente di rappresentare A usando uno spazio in memoria $O(m)$, dove m è il numero di celle di A che non contengono il riferimento `null`. Descrivere modi efficienti per realizzare i metodi dell'ADT "lista con indice" usando tale rappresentazione e farne l'analisi delle prestazioni.
- C-7.48 Progettare un ADT "lista posizionale circolare" che astragga il comportamento di una lista concatenata circolare in modo analogo a quanto l'ADT "lista posizionale" fa con la lista doppiamente concatenata.
- C-7.49 Fornire un'implementazione del metodo `listIterator()`, nel contesto della classe `LinkedPositionalList`, che restituiscia un oggetto che implementi l'interfaccia `java.util.ListIterator` descritta nel Paragrafo 7.5.1.
- C-7.50 Descrivere una strategia per creare iteratori di lista che *falliscano rapidamente (fail fast)*, cioè che diventino tutti non operativi non appena la lista su cui agiscono viene modificata.
- C-7.51 Esiste un semplice algoritmo, denominato *ordinamento a bolle (bubble sort)*, per ordinare una lista L di n elementi confrontabili. Questo algoritmo percorre la lista $n - 1$ volte e, durante ciascuna scansione, confronta l'elemento corrente con il successivo, scambiandoli se sono fuori ordine tra loro. Fornire una descrizione mediante pseudocodice dell'algoritmo di ordinamento a bolle che sia tanto efficiente quanto è possibile, ipotizzando che L sia implementata mediante una lista doppiamente concatenata. Qual è il tempo d'esecuzione di questo algoritmo?
- C-7.52 Ripetere l'esercizio precedente ipotizzando che L sia una lista con indice.
- C-7.53 Descrivere un modo efficiente per gestire una lista di preferiti, L , usando l'euristica *move-to-front*, in modo che gli elementi mai ispezionati negli ultimi n accessi vengano automaticamente eliminati dalla lista.
- C-7.54 Considerando una lista L avente n elementi, gestita in base all'euristica *move-to-front*, descrivere una sequenza di n^2 accessi a L che venga eseguita in un tempo $\Omega(n^3)$.
- C-7.55 Nelle basi di dati (*database*), un'utile operazione è la *fusione naturale (natural join)*. Se immaginiamo che una base di dati sia una lista di coppie *ordinate* di oggetti, allora la fusione naturale delle basi di dati A e B è la lista di tutte le triplette ordinate (x, y, z) tali che la coppia (x, y) appartenga a A e la coppia (y, z) appartenga a B . Descrivere e analizzare un algoritmo efficiente per calcolare la fusione naturale tra la lista A contenente n coppie e la lista B contenente m coppie.
- C-7.56 Quando Bob vuole inviare ad Alice un messaggio M tramite Internet, scomponne M in n pacchetti di dati (*data packet*), li numera consecutivamente e li trasmette. Quando i pacchetti giungono al computer di Alice, possono essere fuori ordine, per cui Alice deve ricostruire la sequenza di n pacchetti nell'ordine corretto prima di poter avere la certezza di aver ricevuto l'intero messaggio. Descrivere una

strategia efficiente perché Alice possa risolvere questo problema. Qual è il tempo d'esecuzione dell'algoritmo?

C-7.57 Implementare la classe `FavoritesList` usando una lista con indice.

Progettazione

P-7.58 Usando tecniche simili a quelle viste nel Paragrafo 4.1, sviluppare un esperimento che verifichi l'efficienza di n invocazioni consecutive del metodo `add` di un esempio di `ArrayList`, al variare di n , in ciascuno dei seguenti scenari:

- Ogni invocazione riguarda l'indice 0.
- Ogni invocazione riguarda l'indice `size()/2`.
- Ogni invocazione riguarda l'indice `size()`.

Analizzare i risultati sperimentali ottenuti.

P-7.58 Modificare la classe `LinkedPositionalList` in modo che si ottenga una segnalazione d'errore in seguito all'utilizzo di una posizione non valida, secondo lo scenario descritto nell'Esercizio R-7.14.

P-7.59 Implementare una classe `CardHand` che rappresenti la disposizione di carte da gioco nella mano di un giocatore. Il simulatore deve rappresentare la sequenza di carte usando un'unica lista posizionale, con le carte dello stesso seme che stanno vicine. Implementare questa strategia pensando a quattro "dita" della mano, una per ciascuno dei semi (cuori, quadri, fiori e picche, rispettivamente *hearts*, *diamonds*, *clubs* e *spades*), in modo che l'aggiunta o la rimozione di una carta dalla mano di un giocatore possa avvenire in un tempo costante. La classe deve mettere a disposizione i metodi seguenti:

- `addCard(r, s)`: Aggiunge alla mano una nuova carta con valore (*rank*) r e seme (*suit*) s .
- `play(s)`: Elimina dalla mano e restituisce una carta del seme s , scelta a caso; se non sono presenti carte del seme s , elimina e restituisce una carta scelta a caso dalla mano.
- `iterator()`: Restituisce un iteratore per tutte le carte della mano.
- `suitIterator(s)`: Restituisce un iteratore per tutte le carte della mano che appartengono al seme s .

P-7.60 Scrivere un semplice editor di testi che memorizzi e visualizzi una sequenza di caratteri usando l'ADT lista posizionale, con un oggetto che funga da cursore e che evidenzi una posizione nella sequenza. L'editor deve mettere a disposizione dell'utente le seguenti operazioni:

- `left`: Sposta il cursore di un carattere verso sinistra (senza far nulla se è già all'inizio).
- `right`: Sposta il cursore di un carattere verso destra (senza far nulla se è già alla fine).

- **insert c:** Inserisce il carattere *c* subito dopo il cursore.
- **delete:** Cancella il carattere che segue immediatamente il cursore (senza far nulla se il cursore è alla fine).

Note

Il modello che considera le strutture dati come raccolte o collezioni si trova nei libri di progettazione orientata agli oggetti di Booch [16], Budd [19] e Liskov e Guttag [67], insieme con altri principi di progettazione orientata agli oggetti. I concetti di lista e iteratore pervadono il Java Collections Framework. Il nostro ADT “lista posizionale” è derivato dall’astrazione di “posizione” introdotta da Aho, Hopcroft e Ullman [6], oltre che nell’ADT “lista” di Wood [96]. Le implementazioni di liste mediante array e liste concatenate sono state discusse da Knuth [60].

8

Alberi

8.1 Alberi generici

Gli esperti di produttività affermano che le vere svolte avvengono pensando “in modo non lineare”. In questo capitolo parleremo di una delle strutture dati non lineari più importanti nel mondo dell’elaborazione: gli *alberi* (*trees*). Le strutture ad albero segnano effettivamente una svolta nell’organizzazione dei dati, perché ci consentono di implementare molti algoritmi in modo che la loro efficienza sia decisamente superiore a quella ottenibile usando strutture dati lineari, come gli array o le liste concatenate. Inoltre, gli alberi costituiscono una scelta naturale per l’organizzazione di molti dati e, di conseguenza, sono ampiamente diffusi nella gestione di *file system*, interfacce grafiche per l’interazione con l’utente, basi di dati, siti web e molti altri sistemi di elaborazione.

Non sempre è chiaro cosa intendano gli esperti di produttività con l’espressione “pensare in modo non lineare”, ma quando diciamo che gli alberi non sono lineari ci riferiamo al fatto che la relazione che organizza i dati all’interno della struttura è più ricca della semplice relazione “prima” e “dopo” che caratterizza oggetti disposti in sequenza. In un albero, le relazioni sono *gerarchiche* (*hierarchical*), con alcuni oggetti che si trovano “sopra” o “sotto” altri oggetti. In realtà, la terminologia maggiormente utilizzata per le strutture dati ad albero deriva dagli alberi genealogici, dato che le parole più comunemente utilizzate per descrivere relazioni sono “genitore” (*parent*), “figlio” (*child*), “antenato” (*ancestor*) e “discendente” (*descendant*). Nella Figura 8.1 riportiamo un esempio di albero genealogico.

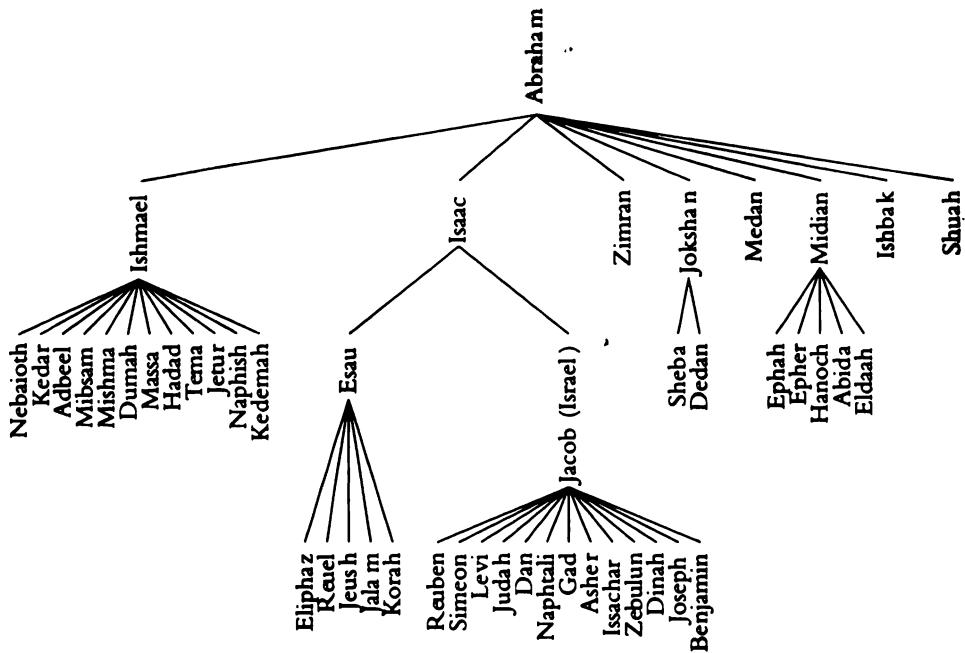


Figura 8.1: Un albero genealogico che mostra alcuni discendenti di Abramo, così come riportato nella Genesi, capitoli 25-36.

8.1.1 Alberi: definizioni e proprietà

Un *albero* (*tree*) è un tipo di dato astratto che memorizza elementi in modo gerarchico. Con l'eccezione dell'elemento che sta in cima, ogni elemento ha un elemento *genitore* (*parent*) e zero o più elementi *figli* (*children*). Solitamente un albero viene visualizzato posizionando gli elementi all'interno di ovali o rettangoli e disegnando segmenti per rappresentare le connessioni tra genitori e figli (come nella Figura 8.2). L'elemento che sta in cima alla gerarchia è la *radice* (*root*) dell'albero e viene disegnato più in alto degli altri elementi, connessi al di sotto (esattamente l'opposto di quanto avviene con un albero nella vita reale).

Definizione di albero

Formalmente, definiamo l'*albero* T come un insieme di *nodi* che memorizzano elementi, in modo che tra i nodi esista una relazione di tipo *genitore-figlio* che soddisfi le seguenti proprietà:

- Se T non è vuoto, ha un nodo speciale, chiamato *radice* di T , che non ha genitore.
- Ciascun nodo v di T che non sia la radice ha un unico nodo *genitore* w ; ogni nodo che abbia w come genitore è un *figlio* di w .

Come si può notare, in base a questa definizione un albero può essere vuoto, nel senso che non ha nessun nodo. Questa convenzione ci consente di definire un albero anche ricorsivamente: un albero T è vuoto oppure è costituito da un nodo r , detto radice di T , e da un insieme (eventualmente vuoto) di sottoalberi le cui radici sono i figli di r .

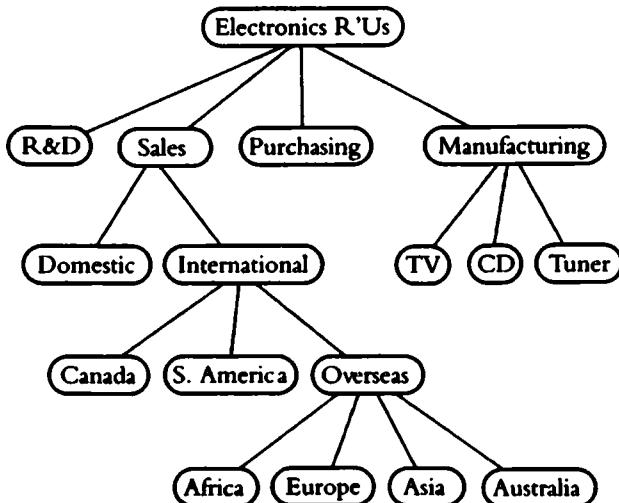


Figura 8.2: Un albero con 17 nodi che rappresenta l'organizzazione di una finta azienda. La radice contiene il nome dell'azienda: *Electronics R'Us*. I figli della radice memorizzano i nomi dei settori principali dell'azienda: *R&D*, *Sales*, *Purchasing* e *Manufacturing*. I nodi interni dell'albero sono quelli che contengono *Sales*, *International*, *Overseas*, *Electronics R'Us* e *Manufacturing*.

Altre relazioni tra i nodi

Due nodi che siano figli di uno stesso genitore si dicono *fratelli (siblings)*. Un nodo v è *esterno (external)* se non ha figli, mentre è *interno (internal)* se ha uno o più figli. I nodi esterni prendono anche il nome di *foglie (leaf, plurale leaves)*.

Esempio 8.1: Nel Paragrafo 5.1.4 abbiamo visto la relazione gerarchica esistente tra i file e le cartelle nel file system di un calcolatore, anche se in quel caso non abbiamo posto l'accento sulla nomenclatura di un file system visto come albero. Nella Figura 8.3 rivediamo un esempio già trattato: i nodi interni dell'albero sono associati alle cartelle, mentre le foglie rappresentano i file. Nei sistemi operativi Unix e Linux, la radice dell'albero viene proprio chiamata "cartella radice" (root directory) e viene rappresentata dal simbolo "/".

Un nodo u è un *antenato* di un nodo v se $u = v$ oppure se u è un antenato del genitore di v . Viceversa, diciamo che un nodo v è un *discendente* di un nodo u se u è un antenato di v . Ad esempio, nella Figura 8.3, `cs252/` è un antenato di `papers/` e `pr3` è un discendente di `cs016/`. Il *sottoalbero (subtree)* di T avente radice nel nodo v è l'albero costituito da tutti i discendenti di v che appartengono a T (compreso v stesso). Nella Figura 8.3, il sottoalbero avente radice `cs016/` è costituito dai nodi `cs016/`, `grades/`, `homeworks/`, `programs/`, `hw1`, `hw2`, `hw3`, `pr1`, `pr2` e `pr3`.

Rami e percorsi negli alberi

Un *ramo (edge)* di un albero T è una coppia di nodi (u, v) tali che u è il genitore di v , o viceversa. Un *percorso (path)* di T è una sequenza di nodi tale che ciascuna coppia di nodi consecutivi nella sequenza sia un ramo. Ad esempio, l'albero della Figura 8.3 contiene il percorso `(cs252/, projects/, demos/, market)`.

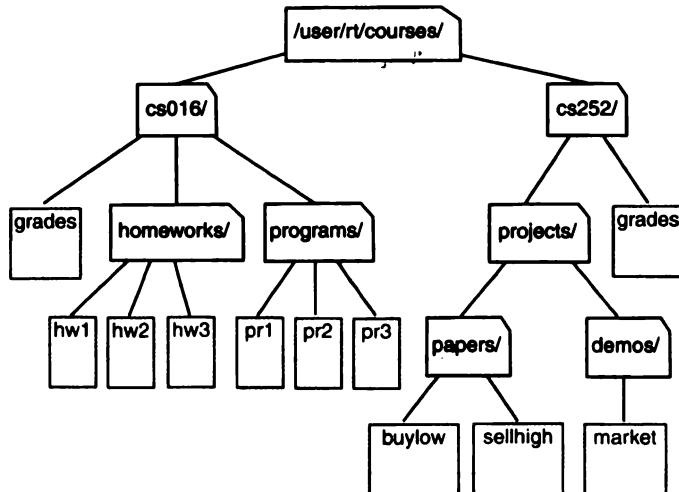


Figura 8.3: Un albero che rappresenta una porzione di un file system.

Alberi ordinati

Un albero è *ordinato* (*ordered*, e non *sorted*, che è una traduzione di “ordinato” con un significato diverso) se esiste una disposizione lineare significativa che coinvolga i figli di ciascun nodo, cioè se ha senso, per i dati rappresentati, identificare i figli di un nodo come “primo figlio”, “secondo figlio” e così via. Tale ordinamento posizionale viene solitamente visualizzato disponendo i fratelli da sinistra a destra, secondo il loro ordinamento (il primo nodo è quello più a sinistra).

Esempio 8.2: Gli elementi che compongono un documento strutturato, come un libro, sono organizzati gerarchicamente come un albero, i cui nodi interni sono le parti, i capitoli e le sezioni del libro, mentre le foglie sono i paragrafi di testo, le tabelle, le figure e così via (come si può vedere nella Figura 8.4). La radice dell’albero corrisponde all’intero libro. Potremmo, in effetti, immaginare di espandere ulteriormente l’albero, facendo vedere che i paragrafi sono composti da frasi, che le frasi sono composte di parole e che le parole sono costituite da caratteri. Un tale albero è un esempio di albero ordinato, perché tra i figli di ciascun nodo esiste un ordinamento ben definito.

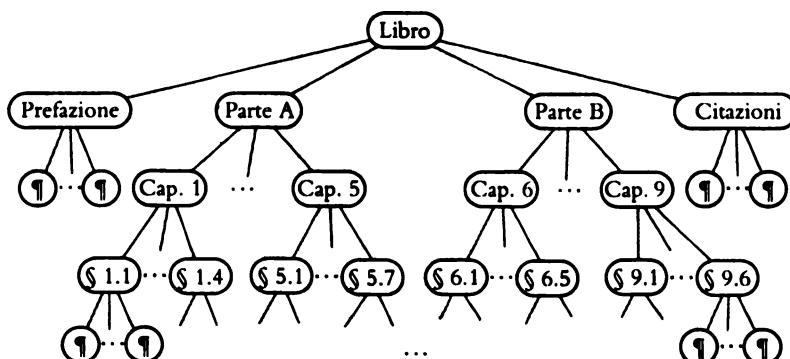


Figura 8.4: Un albero ordinato associato a un libro.

Torniamo a esaminare gli altri esempi di alberi già descritti e vediamo se l'ordine tra i figli è rilevante. Un albero genealogico che descrive relazioni tra le generazioni, come nella Figura 8.1, viene spesso rappresentato come albero ordinato, con i fratelli disposti in base alla data di nascita.

Al contrario, il diagramma organizzativo di un'azienda, come quello riportato nella Figura 8.2, viene solitamente considerato come albero non ordinato. Analogamente, quando usiamo un albero per descrivere una gerarchia di ereditarietà tra classi, come nella Figura 2.7, non ha particolarmente senso ordinare le sottoclassi di una stessa superclasse genitore. Infine, analizziamo l'uso di un albero per rappresentare un file system in un calcolatore, come nella Figura 8.3: sebbene spesso i sistemi operativi visualizzino le entità (file e cartelle) presenti all'interno di una particolare cartella secondo un ordine ben determinato (ad esempio, alfabetico o cronologico), tipicamente non si tratta di un ordinamento che attiene intrinsecamente alla rappresentazione del file system.

8.1.2 L'albero come tipo di dato astratto

Come già fatto nel Paragrafo 7.3 in merito alle liste posizionali, definiamo il tipo di dato astratto “albero” usando il concetto di *posizione* come astrazione di un nodo dell’albero. In ogni posizione è memorizzato un elemento e le posizioni soddisfano le relazioni genitore-figlio che definiscono la struttura dell’albero. Un oggetto che rappresenta una posizione in un albero mette a disposizione i metodi seguenti:

`getElement()`: Restituisce l’elemento memorizzato in questa posizione.

Il tipo di dato astratto “albero” mette, invece, a disposizione i seguenti *metodi d’accesso*, che consentono a un utilizzatore di navigare tra le diverse posizioni di un albero T :

`root()`: Restituisce la posizione della radice dell’albero (oppure `null` se è vuoto).

`parent(p)`: Restituisce la posizione del genitore della posizione p (oppure `null` se p è la radice).

`children(p)`: Restituisce un contenitore iterabile con i figli della posizione p (se ce ne sono).

`numChildren(p)`: Restituisce il numero di figli della posizione p .

Se T è un albero ordinato, il contenitore restituito da `children(p)` contiene i figli di p in ordine.

Oltre ai fondamentali metodi d’accesso appena descritti, un albero dispone anche dei seguenti *metodi di interrogazione (query methods)*:

`isInternal(p)`: Restituisce `true` se e solo se la posizione p ha almeno un figlio.

`isExternal(p)`: Restituisce `true` se e solo se la posizione p non ha figli.

`isRoot(p)`: Restituisce `true` se e solo se la posizione p è la radice dell’albero.

Questi ultimi metodi rendono semplice e comprensibile la programmazione di algoritmi che usano alberi, perché li possiamo, in particolare, utilizzare nelle espressioni condizionali degli enunciati `if` e `while`.

Inoltre, gli alberi dispongono anche di un certo numero di metodi più generali, non correlati con la struttura specifica degli alberi:

- `size()`: Restituisce il numero di posizioni (e, quindi, di elementi) presenti nell'albero.
- `isEmpty()`: Restituisce `true` se e solo se l'albero non contiene posizioni (e, quindi, non contiene elementi), cioè è vuoto.
- `iterator()`: Restituisce un iteratore che scandisce tutti gli elementi dell'albero (in modo che l'albero stesso sia `Iterable`).
- `positions()`: Restituisce un contenitore iterabile con tutte le posizioni dell'albero.

Se a un metodo qualsiasi viene passata come parametro una posizione non valida, viene lanciata un'eccezione di tipo `IllegalArgumentException`.

In questo momento non definiamo metodi per creare o modificare alberi: preferiamo descrivere tali metodi più avanti e ne vedremo versioni diverse, connesse a specifiche implementazioni dell'interfaccia "albero" e a specifiche applicazioni di alberi.

Un'interfaccia "albero" in Java

Nel Codice 8.1 definiamo in modo formale l'ADT albero mediante l'interfaccia `Tree` in Java. Usiamo la stessa definizione dell'interfaccia `Position` che abbiamo presentato nel Paragrafo 7.3.2 quando abbiamo parlato di liste posizionali. Si noti che dichiariamo l'interfaccia `Tree` in modo che estenda l'interfaccia `Iterable` (e inseriamo anche la dichiarazione del metodo `iterator()`, che è richiesto per implementare `Iterable`).

Codice 8.1: Definizione dell'interfaccia `Tree`.

```

1  /** Un albero i cui nodi possono avere un numero di figli arbitrario. */
2  public interface Tree<E> extends Iterable<E> {
3      Position<E> root();
4      Position<E> parent(Position<E> p) throws IllegalArgumentException;
5      Iterable<Position<E>> children(Position<E> p)
6          throws IllegalArgumentException;
7      int numChildren(Position<E> p) throws IllegalArgumentException;
8      boolean isInternal(Position<E> p) throws IllegalArgumentException;
9      boolean isExternal(Position<E> p) throws IllegalArgumentException;
10     boolean isRoot(Position<E> p) throws IllegalArgumentException;
11     int size();
12     boolean isEmpty();
13     Iterator<E> iterator();
14     Iterable<Position<E>> positions();
15 }
```

La classe di base `AbstractTree` in Java

Nel Paragrafo 2.3 abbiamo discusso il ruolo delle interfacce e delle classi astratte in Java: un'interfaccia è la definizione di un tipo di dati che contiene dichiarazioni pubbliche di

metodi, ma non può includere definizioni del corpo di tali metodi. Al contrario, una *classe astratta* può definire implementazioni concrete di alcuni dei propri metodi, pur lasciando astratti altri metodi, senza definirne il corpo.

Una classe astratta viene progettata per essere, tramite l'ereditarietà, una classe di base per una o più implementazioni concrete di un'interfaccia. Quando alcune funzionalità di un'interfaccia sono implementate in una classe astratta, rimane meno lavoro da compiere per arrivare a una completa implementazione concreta. La libreria standard di Java contiene molte classi astratte, alcune delle quali si trovano nel Java Collections Framework. Per rendere evidente questo obiettivo, tali classi hanno convenzionalmente il nome che inizia con la parola *Abstract*. Ad esempio, c'è una classe, *AbstractCollection*, che implementa alcune delle funzionalità dell'interfaccia *Collection*, un'altra classe, *AbstractQueue*, che implementa alcune delle funzionalità dell'interfaccia *Queue* e, come ultimo esempio, c'è una classe, *AbstractList*, che implementa parte delle funzionalità dell'interfaccia *List*.

Nel caso della nostra interfaccia *Tree*, definiremo una classe di base, *AbstractTree*, che ci sarà utile per mettere in evidenza come molti algoritmi basati su alberi possano essere descritti in modo indipendente dalla rappresentazione di basso livello di una struttura dati ad albero. Infatti, se un'implementazione concreta fornisce tre metodi fondamentali, cioè i metodi *root()*, *parent(p)* e *children(p)*, tutti gli altri comportamenti dell'interfaccia *Tree* possono essere ereditati da quelli della classe *AbstractTree*.

Il Codice 8.2 presenta una prima implementazione della classe *AbstractTree*, che definisce i metodi più banali dell'interfaccia *Tree*. Rimandiamo al Paragrafo 8.4 la discussione sugli algoritmi generali di attraversamento di un albero che possono essere utilizzati per generare il contenitore restituito da *positions()* e utilizzato all'interno della classe *AbstractTree*. Come nel nostro ADT "lista posizionale" visto nel Capitolo 7, la scansione delle *posizioni* di un albero può essere agevolmente adattata per effettuare una scansione degli *elementi* dell'albero, o anche per determinare la dimensione di un albero (anche se le nostre implementazioni concrete di albero proporranno meccanismi più diretti per ottenerne la dimensione).

Codice 8.2: Una prima implementazione della classe di base *AbstractTree* (nei prossimi paragrafi aggiungeremo ulteriori funzionalità a questa classe).

```

1  /** Una classe di base astratta che implementa in parte l'interfaccia Tree. */
2  public abstract class AbstractTree<E> implements Tree<E> {
3      public boolean isInternal(Position<E> p) { return numChildren(p) > 0; }
4      public boolean isExternal(Position<E> p) { return numChildren(p) == 0; }
5      public boolean isRoot(Position<E> p) { return p == root(); }
6      public boolean isEmpty() { return size() == 0; }
7  }
```

8.1.3 Calcolare profondità e altezza

Sia *p* una posizione all'interno dell'albero *T*. La *profondità* (*depth*) di *p* è il numero di antenati di *p*, diversi dalla posizione *p* stessa. Ad esempio, nell'albero della Figura 8.2, il nodo che contiene *International* ha profondità 2. Si osservi che questa definizione implica che la profondità della radice di *T* sia zero. La profondità di *p* può essere definita anche in modo ricorsivo:

- Se p è la radice, allora la profondità di p è 0.
- Altrimenti, la profondità di p è uno più la profondità del genitore di p .

Sulla base di questa definizione, nel Codice 8.3 presentiamo un semplice algoritmo ricorsivo, `depth`, che calcola la profondità di una posizione p nell'albero T . Questo metodo invoca se stesso ricorsivamente usando come parametro il genitore di p , per poi aggiungere 1 al valore ottenuto.

Codice 8.3: Il metodo `depth` implementato nella classe `AbstractTree`.

```

1  /** Restituisce la profondità della Position<E> p. */
2  public int depth(Position<E> p) {
3      if (isRoot(p))
4          return 0;
5      else
6          return 1 + depth(parent(p));
7  }

```

Il tempo d'esecuzione di `depth(p)` per la posizione p è $O(d_p + 1)$, dove d_p indica la profondità di p nell'albero, perché l'algoritmo esegue un passo ricorsivo tempo-costante per ogni antenato di p . Di conseguenza, l'algoritmo `depth(p)` viene eseguito in un tempo $O(n)$ nel caso peggiore, essendo n il numero totale di posizioni in T , perché una posizione di T può avere al massimo profondità $n - 1$, nel caso in cui tutti i nodi interni abbiano un unico figlio. Anche se quest'ultimo tempo d'esecuzione è una funzione della dimensione dell'albero, la prima espressione, che caratterizza il tempo in funzione di d_p , è decisamente più informativa, perché il parametro d_p può essere molto minore di n .

Altezza

Definiamo ora l'*altezza* di un albero come il valore massimo delle profondità delle sue posizioni (oppure zero se l'albero è vuoto). Ad esempio, l'albero della Figura 8.2 ha altezza 4, perché il nodo che contiene *Africa* ha profondità 4, così come i suoi fratelli. È facile osservare che la posizione avente profondità massima deve essere una foglia.

Nel Codice 8.4 presentiamo un metodo che calcola l'altezza di un albero sulla base di questa definizione. Sfortunatamente, tale approccio non è molto efficiente, per cui abbiamo dato all'algoritmo il nome `heightBad` (*bad* significa "cattivo"...) e l'abbiamo dichiarato come metodo privato della classe `AbstractTree` (in modo che non possa essere usato dagli utilizzatori della classe).

Codice 8.4: Il metodo `heightBad` implementato nella classe `AbstractTree`. Si osservi che questo metodo invoca il metodo `depth`, definito nel Codice 8.3.

```

1  /** Restituisce l'altezza dell'albero. */
2  private int heightBad() { // funziona, ma in un tempo quadratico nel caso peggiore
3      int h = 0;
4      for (Position<E> p : positions())
5          if (isExternal(p)) // considera soltanto le posizioni delle foglie
6              h = Math.max(h, depth(p));
7      return h;
8  }

```

Anche se non abbiamo ancora definito il metodo `positions()`, vedremo che lo si può implementare in modo che l'intera scansione delle posizioni di un albero T avvenga in un tempo $O(n)$, essendo n il numero di posizioni di T . Dato che `heightBad` invoca l'algoritmo `depth(p)` per ciascuna foglia di T , il suo tempo d'esecuzione è $O(n + \sum_{p \in L} (d_p + 1))$, dove L è l'insieme delle posizioni di T che siano foglie. Nel caso peggiore, tale sommatoria ha un valore proporzionale a n^2 (si veda l'Esercizio C-8.31), per cui l'algoritmo `heightBad` viene eseguito, nel caso peggiore, in un tempo $O(n^2)$.

L'altezza di un albero può essere calcolata in modo più efficiente, in un tempo $O(n)$ nel caso peggiore, considerando una sua definizione ricorsiva. Per farlo, descriveremo una funzione che usa come parametro una posizione dell'albero e calcola l'altezza del sottoalbero avente radice in tale posizione. Dal punto di vista formale, definiamo l'*altezza* di una posizione p nell'albero T in questo modo:

- Se p è una foglia, allora l'altezza di p è 0.
- Altrimenti, l'altezza di p è uno più il valore massimo delle altezze dei figli di p .

La proposizione seguente mette in relazione la nostra prima definizione di altezza di un albero con l'altezza della posizione *radice*, usando la formula ricorsiva appena definita.

Proposizione 8.3: *L'altezza della radice di un albero non vuoto T , calcolata usando la definizione ricorsiva, è uguale al valore massimo delle profondità delle foglie di T .*

L'Esercizio R-8.3 si occuperà di dimostrare questa proposizione.

Il Codice 8.5 riporta un'implementazione dell'algoritmo ricorsivo che calcola l'altezza di un sottoalbero avente radice in una data posizione, p . L'altezza di un albero non vuoto può essere calcolata fornendo semplicemente la radice dell'albero come parametro del metodo.

Codice 8.5: Il metodo `height` per calcolare l'altezza del sottoalbero avente radice nella posizione p di un `AbstractTree`.

```

1  /** Restituisce l'altezza del sottoalbero avente radice nella Position p. */
2  public int height(Position<E> p) {
3      int h = 0;          // caso base se p è una posizione esterna, cioè una foglia
4      for (Position<E> c : children(p))
5          h = Math.max(h, 1 + height(c));
6      return h;
7  }

```

È importante capire quale sia il motivo che rende il metodo `height` più efficiente del metodo `heightBad`. L'algoritmo è ricorsivo e si muove dall'alto verso il basso. Se il metodo viene invocato inizialmente ricevendo la radice di T , prima o poi verrà invocato una volta per ciascuna posizione di T , perché la radice invoca la ricorsione su ciascuno dei suoi figli, i quali a loro volta invocheranno la ricorsione su ciascuno dei loro figli, e così via.

Per determinare il tempo d'esecuzione dell'algoritmo `height` ricorsivo possiamo sommare, per tutte le posizioni, la quantità di tempo spesa per eseguire la parte non ricorsiva di ciascuna invocazione (ricordando il Paragrafo 5.2 dove abbiamo visto come analizzare i processi ricorsivi). Nella nostra implementazione, per ciascuna posizione è richiesto un

lavoro costante, a cui si somma il tempo richiesto per calcolare il valore massimo durante la scansione dei figli. Pur non disponendo ancora di un'implementazione concreta di $\text{children}(p)$, ipotizziamo che tale scansione venga eseguita in un tempo $O(c_p + 1)$, dove c_p indica il numero di figli di p . L'algoritmo $\text{height}(p)$ dedica un tempo $O(c_p + 1)$ al calcolo del massimo in ciascuna posizione p , quindi il tempo d'esecuzione totale è $O(\sum_p (c_p + 1)) = O(n + \sum_p c_p)$. Per completare l'analisi, usiamo la proprietà che segue.

Proposizione 8.4: Sia T un albero avente n posizioni e c_p indichi il numero di figli della posizione p di T . Sommando su tutte le posizioni di T , si ottiene $\sum_p c_p = n - 1$.

Dimostrazione: Ciascuna posizione di T , tranne la radice, è figlia di un'altra posizione, quindi il suo contributo alla sommatoria è pari a un'unità. ■

Per la Proposizione 8.4, il tempo d'esecuzione dell'algoritmo height , quando viene invocato con la radice di T , è $O(n)$, dove n è il numero di posizioni di T .

8.2 Alberi binari

Un *albero binario (binary tree)* è un albero ordinato avente le seguenti proprietà:

1. Ogni nodo ha al massimo due figli.
2. Ogni nodo figlio è etichettato come *figlio sinistro (left child)* o come *figlio destro (right child)*.
3. Il figlio sinistro precede il figlio destro nell'ordinamento tra i figli di un nodo.

Il sottoalbero avente radice nel figlio sinistro o destro di un nodo interno v viene chiamato *sottoalbero sinistro* o, rispettivamente, *sottoalbero destro* di v . Un albero binario si dice *proprio (proper)* se ogni suo nodo ha zero o due figli; un tale albero viene anche, a volte, chiamato *pieno (full)*. Quindi, in un albero binario proprio, ogni nodo interno ha esattamente due figli. Un albero binario che non sia proprio si dice *improprio (improper)*.

Esempio 8.5: Una categoria importante di alberi binari viene usata in quei contesti nei quali si vuole rappresentare un certo numero di risultati diversi che si possono produrre in seguito alle risposte a una serie di domande di tipo sì/no. Ogni nodo interno viene associato a una domanda. Partendo dalla radice, procediamo verso il figlio sinistro o destro del nodo in esame in relazione al fatto che la risposta alla domanda sia "sì" oppure "no", quindi, ogni volta che prendiamo una decisione, seguiamo un ramo che va da un genitore a un suo figlio, generando così un percorso nell'albero che va dalla radice a una foglia. Gli alberi binari di questo tipo si chiamano *alberi di decisione (decision tree)*, perché una foglia p di un tale albero rappresenta una decisione in merito a cosa fare se alle domande associate agli antenati di p sono state fornite risposte che hanno portato, appunto, in p . Un albero di decisione è un albero binario proprio e la Figura 8.5 mostra un albero di decisione che fornisce suggerimenti a un futuro investitore finanziario.

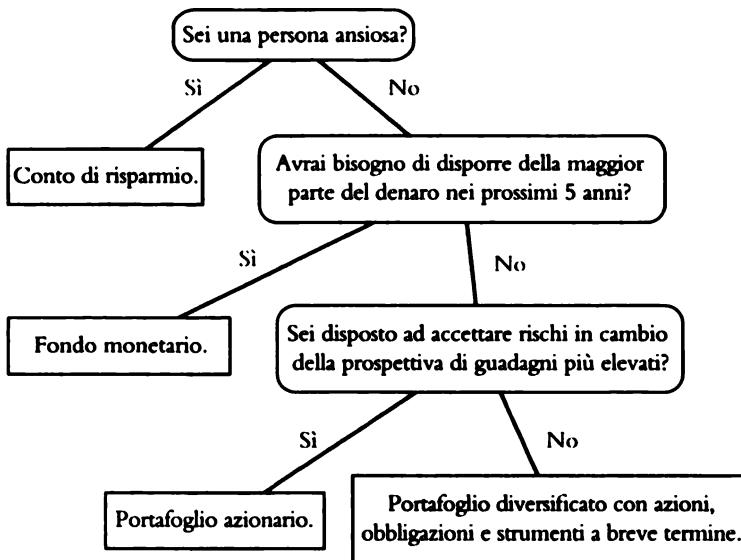


Figura 8.5: Un albero di decisione che fornisce suggerimenti per effettuare un investimento finanziario.

Esempio 8.6: Un'espressione aritmetica può essere rappresentata da un albero binario le cui foglie sono associate a variabili e costanti, mentre i nodi interni sono associati a uno degli operatori $+$, $-$, \times e $/$, come si può vedere nella Figura 8.6. In un tale albero ogni nodo è associato a un valore:

- Se il nodo è una foglia, il suo valore è quello della sua variabile o costante.
- Se il nodo è interno, il suo valore è definito applicando il suo operatore ai valori dei suoi figli.

Un albero che rappresenta in questo modo un'espressione aritmetica è solitamente un albero binario proprio, perché tutti gli operatori considerati, $+$, $-$, \times e $/$, richiedono proprio due operandi. Naturalmente, se volessimo consentire la presenza nell'espressione di operatori unari, come la negazione ($-$, come in " $-x$ "), allora l'albero binario potrebbe essere improprio.

Una definizione ricorsiva di albero binario

L'albero binario può essere definito anche in modo ricorsivo. In tal caso, un albero binario è:

- un albero vuoto, oppure
- un albero non vuoto avente il nodo radice r che memorizza un elemento e due alberi binari che sono, rispettivamente, il sottoalbero sinistro e destro di r . Si noti che, in base a questa definizione, uno di tali sottoalberi può essere vuoto, così come possono essere vuoti entrambi.

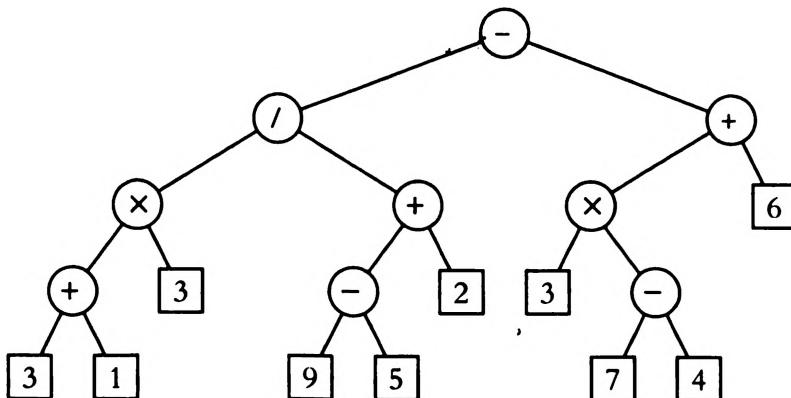


Figura 8.6: Un albero binario che rappresenta un'espressione aritmetica. Questo albero, in particolare, rappresenta l'espressione $((((3 + 1) \times 3) / ((9 - 5) + 2)) - ((3 \times (7 - 4)) + 6))$, quindi, ad esempio, il valore associato al nodo interno che ha "/" come etichetta è 2.

8.2.1 L'albero binario come tipo di dato astratto

Come tipo di dato astratto, l'albero binario è uno speciale albero che fornisce supporto a tre metodi d'accesso aggiuntivi:

- left(*p*):** Restituisce la posizione del figlio sinistro di *p* (oppure `null` se *p* non ha figlio sinistro).
- right(*p*):** Restituisce la posizione del figlio destro di *p* (oppure `null` se *p* non ha figlio destro).
- sibling(*p*):** Restituisce la posizione del fratello di *p* (oppure `null` se *p* non ha fratello).

Proprio come abbiamo fatto nel Paragrafo 8.1.2 per il tipo di dato astratto "albero", non definiamo in questo momento metodi specifici per aggiornare l'albero binario: lo faremo quando descriveremo applicazioni di alberi binari e loro implementazioni specifiche.

Definizione dell'interfaccia `BinaryTree`

Il Codice 8.6 formalizza l'ADT "albero binario" definendo, in Java, l'interfaccia `BinaryTree`, che estende l'interfaccia `Tree` già definita nel Paragrafo 8.1.2 e aggiunge i tre nuovi comportamenti che abbiamo presentato. In questo modo, un albero binario avrà tutte le funzionalità definite per gli alberi generici (ad esempio, `root`, `isExternal`, `parent`, ...) e i nuovi comportamenti `left`, `right` e `sibling`.

Codice 8.6: L'interfaccia `BinaryTree` che estende l'interfaccia `Tree` definita nel Codice 8.1.

```

1  /** Un albero binario. */
2  public interface BinaryTree<E> extends Tree<E> {
3      /** Restituisce la posizione del figlio sinistro di p (o null se non esiste). */
4      Position<E> left(Position<E> p) throws IllegalArgumentException;
5      /** Restituisce la posizione del figlio destro di p (o null se non esiste). */
6      Position<E> right(Position<E> p) throws IllegalArgumentException;

```

```

1  /** Restituisce la posizione del fratello di p (o null se non esiste). */
2  Position<E> sibling(Position<E> p) throws IllegalArgumentException;
3 }

```

Definizione della classe di base astratta AbstractBinaryTree

Proseguiamo con la definizione di classi di base astratte, per migliorare la riutilizzabilità del nostro codice. La classe `AbstractBinaryTree`, presentata nel Codice 8.7, eredita dalla classe `AbstractTree` vista nel Paragrafo 8.1.2, definendo ulteriori metodi concreti che sfruttano i nuovi metodi `left` e `right` (che rimangono astratti).

Il nuovo metodo `sibling` usa una combinazione dei metodi `left`, `right` e `parent`. Di norma il fratello di una posizione `p` viene identificato come l'altro figlio del genitore di `p`, ma `p` non ha un fratello se è la radice o se è l'unico figlio del proprio genitore.

Possiamo usare i metodi `left` e `right`, sapendo che prima o poi verranno certamente definiti, anche per fornire un'implementazione concreta dei metodi `numChildren` e `children`, che fanno parte dell'interfaccia `Tree` originaria. Usando la terminologia del Paragrafo 7.4, l'implementazione del metodo `children` si basa sulla produzione di uno *snapshot*, una fotografia: creiamo un esemplare vuoto di `java.util.ArrayList`, che è un contenitore iterabile, e vi aggiungiamo tutti i figli che esistono, ordinati in modo che il figlio sinistro, se esiste, venga prima di quello destro.

Codice 8.7: Definizione della classe `AbstractBinaryTree` che estende la classe `AbstractTree` definita nel Codice 8.2 e implementa l'interfaccia `BinaryTree` definita nel Codice 8.6.

```

1  /** Una classe di base astratta che implementa in parte l'interfaccia BinaryTree. */
2  public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
3                                     implements BinaryTree<E> {
4      /** Restituisce la posizione del fratello di p (o null se non esiste). */
5      public Position<E> sibling(Position<E> p) {
6          Position<E> parent = parent(p);
7          if (parent == null) return null; // p è la radice
8          if (p == left(parent))           // p è il figlio sinistro
9              return right(parent);      // (può essere null)
10         else                          // p è il figlio destro
11             return left(parent);       // (può essere null)
12     }
13     /** Restituisce il numero di figli della Position p. */
14     public int numChildren(Position<E> p) {
15         int count=0;
16         if (left(p) != null)
17             count++;
18         if (right(p) != null)
19             count++;
20         return count;
21     }
22     /** Restituisce un contenitore iterabile delle posizioni dei figli di p. */
23     public Iterable<Position<E>> children(Position<E> p) {
24         List<Position<E>> snapshot = new ArrayList<>(2); // capacità massima 2
25         if (left(p) != null)
26             snapshot.add(left(p));
27         if (right(p) != null)
28             snapshot.add(right(p));
29         return snapshot;
30     }
31 }

```

8.2.2 Proprietà degli alberi binari

Gli alberi binari hanno alcune proprietà interessanti, che riguardano le relazioni esistenti tra l'altezza dell'albero e il numero dei suoi nodi. Chiamiamo *livello* (*level*) d dell'albero T l'insieme di tutti i nodi di T che hanno la stessa profondità, d . In un albero binario, il livello 0 contiene al massimo un solo nodo (la radice), il livello 1 contiene al massimo due nodi (i figli della radice), il livello 2 al massimo quattro nodi, e così via (come si può vedere nella Figura 8.7). In generale, il livello d ha al massimo 2^d nodi.

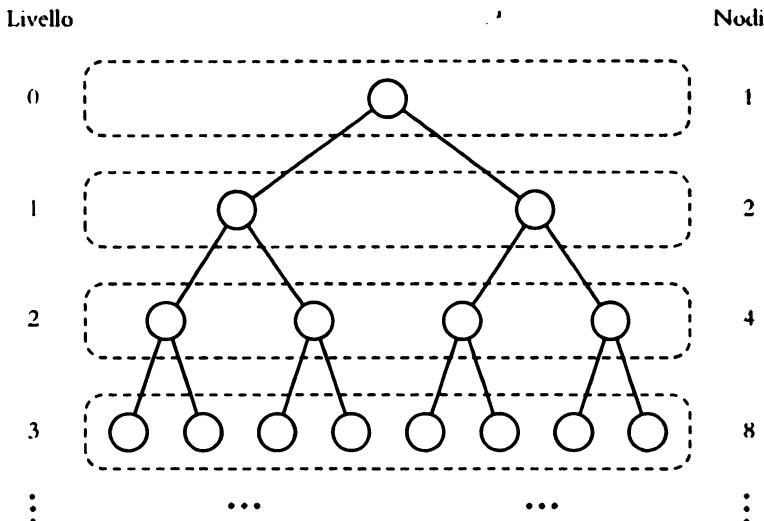


Figura 8.7: Massimo numero di nodi nei livelli di un albero binario.

Come si può notare, il massimo numero di nodi che possono costituire i livelli di un albero binario aumenta esponenzialmente scendendo lungo l'albero. Da questa semplice osservazione, possiamo dedurre alcune proprietà che mettono in relazione l'altezza di un albero binario, T , con il numero dei suoi nodi. La dimostrazione di queste proprietà è rimandata all'Esercizio R-8.8.

Proposizione 8.7: Sia T un albero binario non vuoto e siano n , n_E , n_I e h , rispettivamente, il numero di nodi, il numero di nodi esterni, il numero di nodi interni e l'altezza di T . L'albero T gode delle seguenti proprietà:

1. $h + 1 \leq n \leq 2^{h+1} - 1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq n - 1$

Inoltre, se T è proprio, valgono le seguenti proprietà:

1. $2h + 1 \leq n \leq 2^{h+1} - 1$
2. $h + 1 \leq n_E \leq 2^h$

$$3. h \leq n_I \leq 2^h - 1$$

$$4. \log(n + 1) - 1 \leq h \leq (n - 1)/2$$

Relazione tra nodi interni e nodi esterni in un albero binario proprio

Oltre alle proprietà degli alberi binari appena presentate, tra il numero di nodi interni e il numero di nodi esterni di un albero binario proprio esiste la relazione seguente.

Proposizione 8.8: In un albero binario proprio e non vuoto, T , avente n_E nodi esterni e n_I nodi interni, si ha che $n_E = n_I + 1$.

Dimostrazione: Dimostriamo questa affermazione rimuovendo i nodi da T e suddividendoli in due “mucchietti” inizialmente vuoti, uno per i nodi interni e uno per i nodi esterni, fino a quando T non diventa vuoto. Al termine, dimostreremo che il mucchietto dei nodi esterni contiene un nodo in più rispetto al mucchietto dei nodi interni. Consideriamo due casi:

Caso 1: Se T contiene soltanto il nodo v , rimuoviamo v e lo mettiamo nel mucchietto dei nodi esterni. Quindi, al termine della procedura, il mucchietto dei nodi esterni ha un solo nodo, mentre il mucchietto dei nodi interni è vuoto.

Caso 2: Altrimenti T ha più di un nodo; rimuoviamo da T un nodo esterno qualsiasi, diciamo w , e il suo genitore v , che è ovviamente un nodo interno. Mettiamo, quindi, w nel mucchietto dei nodi esterni e v in quello dei nodi interni. Se v ha un genitore, u , collegiammo u con il nodo z che era fratello di w , come si può vedere nella Figura 8.8. Questa operazione rimuove complessivamente un nodo interno e un nodo esterno, lasciando l’albero ancora binario e proprio.

Ripetendo questa operazione, prima o poi rimarrà un albero costituito da un unico nodo. Si noti che, a quel punto, è stato rimosso lo stesso numero di nodi esterni e di nodi interni, che sono stati posti nel rispettivo mucchietto dalla sequenza di operazioni che ha portato a tale albero finale. Ora, rimuoviamo anche l’ultimo nodo rimasto e lo mettiamo nel mucchietto dei nodi esterni: di conseguenza, il mucchietto dei nodi esterni ha un nodo in più del mucchietto dei nodi interni. ■

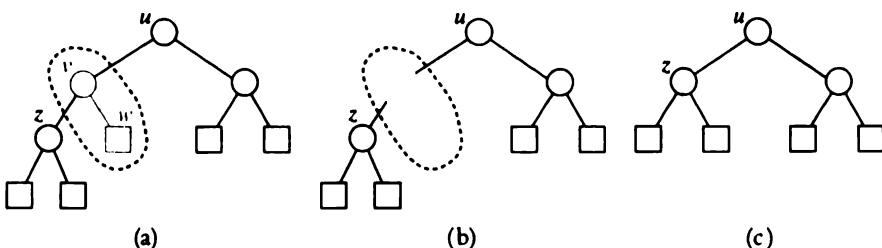


Figura 8.8: Operazione che rimuove un nodo esterno e il suo nodo genitore: è utilizzata per dimostrare la Proposizione 8.8.

Osserviamo che la relazione appena dimostrata non è valida, in generale, per alberi binari impropri e per alberi non binari, anche se in tali casi valgono altre relazioni interessanti (si vedano gli Esercizi C-8.30, C-8.31 e C-8.32).

8.3 Implementare alberi

La classi `AbstractTree` e `AbstractBinaryTree` che abbiamo definito fin qui, in questo capitolo, sono entrambe *classi di base astratte*: anche se mettono già a disposizione molti metodi, non si possono creare esemplari di nessuna delle due. Non abbiamo ancora definito i dettagli più importanti dell'implementazione, relativi alla rappresentazione interna di un albero, né abbiamo deciso come si possa effettivamente navigare nell'albero, spostandosi da un genitore ai suoi figli e viceversa.

Per la rappresentazione di alberi ci sono alcune alternative praticabili, le più diffuse delle quali verranno descritte in questo paragrafo. Iniziamo con il caso di un *albero binario*, perché la sua forma è definita in modo più stringente.

8.3.1 Una struttura concatenata per alberi binari

Una strategia naturale per la realizzazione di un albero binario T è l'uso di una *struttura concatenata*, dove un nodo (come si vede nella Figura 8.9a) associato alla posizione p gestisce riferimenti all'elemento memorizzato nella posizione p e ai nodi associati ai figli e al genitore di p . Se p è la radice di T , allora il campo `parent` di p è `null`. Analogamente, se p non ha figlio sinistro (o, rispettivamente, destro), il campo corrispondente è `null`. L'albero gestisce una variabile di esemplare che memorizza un riferimento al nodo radice (se esiste) e una variabile, detta `size`, che contiene il numero totale di nodi di T . La Figura 8.9b mostra una tale struttura concatenata che rappresenta un albero binario.

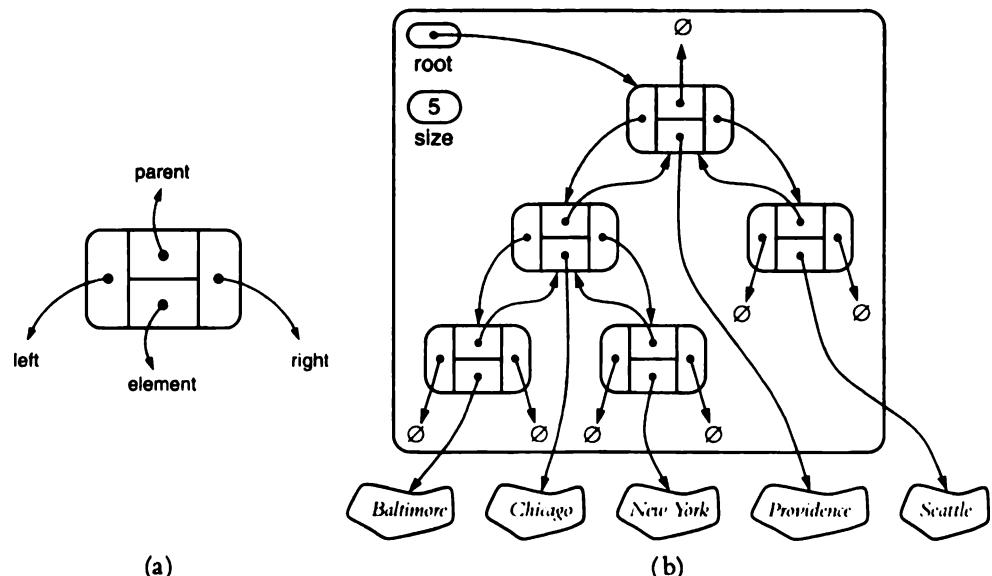


Figura 8.9: Una struttura concatenata che rappresenta: (a) un singolo nodo; (b) un albero binario.

Operazioni per la modifica di un albero binario concatenato

Le interfacce `Tree` e `BinaryTree` definiscono molti metodi per ispezionare un albero già esistente, ma non dichiarano alcun metodo modificatore. Nell'ipotesi, spesso verificata, che un nuovo albero, appena costruito, sia vuoto, ci piacerebbe poter disporre di metodi che ne modifichino la struttura e il contenuto.

Sebbene il principio dell'incapsulamento suggerisca che i comportamenti di un tipo di dato astratto visibili all'esterno non dipendano dalla rappresentazione interna, l'efficienza delle operazioni è fortemente dipendente proprio dalla rappresentazione adottata. Preferiamo, quindi, che ciascuna implementazione concreta di una classe che rappresenta un albero fornisca il supporto più adeguato ai metodi modificatori dell'albero stesso. Nel caso di un albero binario concatenato (cioè realizzato mediante una struttura concatenata), suggeriamo la definizione dei metodi seguenti:

addRoot(*e*): Crea la radice in un albero vuoto, memorizzandovi l'elemento *e*, e restituisce la posizione di tale radice; se l'albero non è vuoto si verifica un errore.

addLeft(*p*, *e*): Crea il figlio sinistro per la posizione *p*, memorizzandovi l'elemento *e*, e restituisce la posizione di tale nuovo nodo; se *p* ha già il figlio sinistro si verifica un errore.

addRight(*p*, *e*): Crea il figlio destro per la posizione *p*, memorizzandovi l'elemento *e*, e restituisce la posizione di tale nuovo nodo; se *p* ha già il figlio destro si verifica un errore.

set(*p*, *e*): Sostituisce con l'elemento *e* l'elemento memorizzato nella posizione *p* e restituisce tale elemento sostituito.

attach(*p*, *T*₁, *T*₂): Collega la struttura interna degli alberi *T*₁ e *T*₂ come, rispettivamente, sottoalbero sinistro e destro della foglia *p*, facendo in modo che *T*₁ e *T*₂ diventino, poi, alberi vuoti; se *p* non è una foglia si verifica un errore.

remove(*p*): Elimina il nodo in posizione *p*, sostituendolo con il suo figlio (se esiste), e restituisce l'elemento precedentemente memorizzato in *p*; se *p* ha due figli si verifica un errore.

Abbiamo scelto con cura proprio questo insieme di operazioni perché ciascuna di esse può essere realizzata in modo che venga eseguita in un tempo $O(1)$ nel caso peggiore quando viene usata la nostra rappresentazione concatenata. I metodi più complessi sono `attach` e `remove`, per via dei vari casi che vanno presi in esame in relazione alle diverse relazioni genitore-figlio e alle situazioni limite, ma rimangono eseguibili mediante un numero costante di operazioni elementari (l'implementazione di entrambi i metodi potrebbe semplificarsi grandemente se usassimo una rappresentazione di albero dotata di nodi sentinella, in modo analogo a quanto fatto per le liste posizionali; si veda l'Esercizio C-8.38).

Implementazione in Java di un albero binario mediante struttura concatenata

Presentiamo ora un'implementazione concreta di una classe, `LinkedBinaryTree`, che realizza il tipo di dato astratto "albero binario" e fornisce supporto per i metodi appena descritti. Questa nuova classe estende la classe di base `AbstractBinaryTree`, ereditando da essa l'imme-

mentazione concreta di alcuni metodi (oltre alla dichiarazione formale di implementazione dell'interfaccia `BinaryTree`).

I dettagli di basso livello della nostra implementazione concatenata di albero ricordano molto da vicino le tecniche usate per realizzare la classe `LinkedPositionalList`, nel Paragrafo 7.3.3. Definiamo una classe annidata non pubblica, `Node`, per rappresentare i nodi, i cui esemplari saranno le posizioni (`Position`) dell'interfaccia pubblica. Come nella Figura 8.9, un nodo gestisce i riferimenti a un elemento, al proprio genitore, al proprio figlio sinistro e al proprio figlio destro (ciascuno dei quali può essere nullo). Un esemplare di albero possiede, invece, un riferimento al proprio nodo radice (eventualmente nullo) e il conteggio dei nodi presenti nell'albero.

Definiamo anche un metodo ausiliario, `validate`, che viene invocato ognqualvolta si riceve come parametro un riferimento di tipo `Position`, per verificare che si tratti di un nodo valido. Nel caso di un albero concatenato, adottiamo la convenzione di assegnare al campo `parent` di un nodo il riferimento a se stesso nel momento in cui il nodo viene eliminato dall'albero a cui appartiene, per poterlo più tardi identificare come posizione non valida.

Le sezioni di Codice 8.8, 8.9, 8.10 e 8.11 presentano interamente la classe `LinkedBinaryTree` e qui delineiamo una breve guida alla comprensione di quel codice.

- Il Codice 8.8 contiene la definizione della classe annidata `Node`, che implementa l'interfaccia `Position`. Questa sezione di codice definisce anche un metodo, `createNode`, che restituisce un nuovo esemplare di nodo, rappresentando così un esempio dello schema progettuale che prende il nome di *factory method pattern* (cioè “metodo-fabbrica”): in questo modo possiamo poi creare sottoclassi che usino un tipo di nodo più specifico (come vedremo nel Paragrafo 11.2.1). La sezione di codice si conclude con la dichiarazione delle variabili di esemplare e del costruttore della classe esterna, `LinkedBinaryTree`.
- Il Codice 8.9 definisce il metodo `protected validate(p)`, seguito dai metodi di accesso `size`, `root`, `parent`, `left` e `right`. Osserviamo che tutti gli altri metodi delle interfacce `Tree` e `BinaryTree` sono definiti in funzione di questi quattro metodi concreti, attraverso le classi di base `AbstractTree` e `AbstractBinaryTree`.
- Il Codice 8.10 e il Codice 8.11 definiscono i sei metodi modificatori di un albero binario concatenato, come descritto in precedenza. Osserviamo che i tre metodi `addRoot`, `addLeft` e `addRight`, creano nuovi esemplari di nodi usando il metodo-fabbrica `createNode`. Il metodo `remove`, presentato a conclusione del Codice 8.11, assegna volutamente al campo `parent` del nodo rimosso un riferimento al nodo stesso, per rispettare la convenzione che abbiamo scelto di adottare per la rappresentazione di un nodo non più valido; agli altri campi riferimento viene assegnato `null`, per agevolare l'azione del `garbage collector`.

Codice 8.8: Un'implementazione della classe `LinkedBinaryTree` (che prosegue nel Codice 8.9, 8.10 e 8.11).

```

1  /** Implementazione concreta di albero binario usando nodi concatenati. */
2  public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {
3
4      //----- classe Node annidata -----
5      protected static class Node<E> implements Position<E> {
6          private E element;           // l'elemento memorizzato in questo nodo
7          private Node<E> parent;    // riferimento al nodo genitore (se esiste)

```

```

8   private Node<E> left;    // riferimento al figlio sinistro (se esiste)
9   private Node<E> right;   // riferimento al figlio destro (se esiste)
10  /** Costruisce un nodo dato un elemento e i nodi adiacenti. */
11  public Node<E> (E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {
12      element = e;
13      parent = above;
14      left = leftChild;
15      right = rightChild;
16  }
17  // metodi d'accesso
18  public E getElement() { return element; }
19  public Node<E> getParent() { return parent; }
20  public Node<E> getLeft() { return left; }
21  public Node<E> getRight() { return right; }
22  // metodi modificatori
23  public void setElement(E e) { element = e; }
24  public void setParent(Node<E> parentNode) { parent = parentNode; }
25  public void setLeft(Node<E> leftChild) { left = leftChild; }
26  public void setRight(Node<E> rightChild) { right = rightChild; }
27 } //---- fine della classe Node annidata -----
28
29  /** Metodo-fabbrica che crea un nuovo nodo memorizzandovi l'elemento e. */
30  protected Node<E> createNode(E e, Node<E> parent,
31                                Node<E> left, Node<E> right) {
32      return new Node<E>(e, parent, left, right);
33  }
34
35  // variabili di esemplare della classe LinkedBinaryTree
36  protected Node<E> root = null; // radice dell'albero
37  private int size = 0;          // numero di nodi dell'albero
38
39  // costruttore
40  public LinkedBinaryTree() { } // costruisce un albero binario vuoto

```

Codice 8.9: Un'implementazione della classe `LinkedBinaryTree` (che continua dal Codice 8.8 e prosegue nel Codice 8.10 e 8.11).

```

41  // metodi ausiliari non pubblici
42  /** Verifica la validità della posizione e la restituisce sotto forma di nodo. */
43  protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
44      if (!(p instanceof Node))
45          throw new IllegalArgumentException("Not valid position type");
46      Node<E> node = (Node<E>) p;    // cast sicuro
47      if (node.getParent() == node) // la nostra convenzione per indicare un ex nodo
48          throw new IllegalArgumentException("p is no longer in the tree");
49      return node;
50  }
51
52  // metodi d'accesso (non già implementati in AbstractBinaryTree)
53  /** Restituisce il numero di nodi presenti nell'albero. */
54  public int size() {
55      return size;
56  }
57
58  /** Restituisce la Position radice dell'albero (o null se l'albero è vuoto). */
59  public Position<E> root() {
60      return root;

```

```

61 }
62
63 /** Restituisce la Position del genitore di p (o null se p è la radice). */
64 public Position<E> parent(Position<E> p) throws IllegalArgumentException {
65     Node<E> node = validate(p);
66     return node.getParent();
67 }
68
69 /** Restituisce la Position del figlio sinistro di p (o null se non c'è). */
70 public Position<E> left(Position<E> p) throws IllegalArgumentException {
71     Node<E> node = validate(p);
72     return node.getLeft();
73 }
74
75 /** Restituisce la Position del figlio destro di p (o null se non c'è). */
76 public Position<E> right(Position<E> p) throws IllegalArgumentException {
77     Node<E> node = validate(p);
78     return node.getRight();
79 }

```

Codice 8.10: Un'implementazione della classe `LinkedBinaryTree` (che continua dal Codice 8.8 e 8.9 e prosegue nel Codice 8.11).

```

80 // metodi modificatori messi a disposizione da questa classe
81 /** Pone e nella radice di un albero vuoto e restituisce la sua nuova Position. */
82 public Position<E> addRoot(E e) throws IllegalStateException {
83     if (!isEmpty()) throw new IllegalStateException("Tree is not empty");
84     root = createNode(e, null, null, null);
85     size = 1;
86     return root;
87 }
88
89 /** Crea il figlio sinistro di p con l'elemento e; ne restituisce la posizione. */
90 public Position<E> addLeft(Position<E> p, E e)
91             throws IllegalArgumentException {
92     Node<E> parent = validate(p);
93     if (parent.getLeft() != null)
94         throw new IllegalArgumentException("p already has a left child");
95     Node<E> child = createNode(e, parent, null, null);
96     parent.setLeft(child);
97     size++;
98     return child;
99 }
100
101 /** Crea il figlio destro di p con l'elemento e; ne restituisce la posizione. */
102 public Position<E> addRight(Position<E> p, E e)
103             throws IllegalArgumentException {
104     Node<E> parent = validate(p);
105     if (parent.getRight() != null)
106         throw new IllegalArgumentException("p already has a right child");
107     Node<E> child = createNode(e, parent, null, null);
108     parent.setRight(child);
109     size++;
110     return child;
111 }
112
113 /** Sostituisce con e l'elemento in p e restituisce l'elemento sostituito. */

```

```
public E set(Position<E> p, E e) throws IllegalArgumentException {
    Node<E> node = validate(p);
    E temp = node.getElement();
    node.setElement(e);
    return temp;
}
```

Codice 8.11: Un'implementazione della classe `LinkedBinaryTree` (che continua dal Codice 8.8, 8.9 e 8.10).

```
/** Collega gli alberi t1 e t2 come sottoalberi sinistro e destro di p, foglia. */
public void attach(Position<E> p, LinkedBinaryTree<E> t1,
                    LinkedBinaryTree<E> t2) throws IllegalArgumentException {
    Node<E> node = validate(p);
    if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
    size += t1.size() + t2.size();
    if (!t1.isEmpty()) { // collega t1 come sottoalbero sinistro di node
        t1.root.setParent(node);
        node.setLeft(t1.root);
        t1.root = null;
        t1.size = 0;
    }
    if (!t2.isEmpty()) { // collega t2 come sottoalbero destro di node
        t2.root.setParent(node);
        node.setRight(t2.root);
        t2.root = null;
        t2.size = 0;
    }
}
/** Elimina il nodo in posizione p e lo sostituisce con il figlio, se c'è. */
public E remove(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    if (numChildren(p) == 2)
        throw new IllegalArgumentException("p has two children");
    Node<E> child = (node.getLeft() != null ? node.getLeft() : node.getRight());
    if (child != null)
        child.setParent(node.getParent()); // il nonno di child ne diventa genitore
    if (node == root)
        root = child; // child diventa la radice
    else {
        Node<E> parent = node.getParent();
        if (node == parent.getLeft())
            parent.setLeft(child);
        else
            parent.setRight(child);
    }
    size--;
    E temp = node.getElement();
    node.setElement(null); // per aiutare il garbage collector
    node.setLeft(null);
    node.setRight(null);
    node.setParent(null); // convenzione per gli ex nodi
    return temp;
}
} //----- fine della classe LinkedBinaryTree -----
```

Prestazioni dell'albero binario implementato con struttura concatenata

Per riassumere l'efficienza delle strutture concatenate, analizziamo qui i tempi d'esecuzione dei metodi della classe `LinkedBinaryTree`, compresi i metodi ereditati da `AbstractTree` e da `AbstractBinaryTree`.

- Il metodo `size` implementato in `LinkedBinaryTree` usa una variabile di esemplare che memorizza il numero di nodi presenti nell'albero e, quindi, viene eseguito in un tempo $O(1)$. Il metodo `isEmpty`, ereditato da `AbstractTree`, si basa su un'unica invocazione di `size` e, quindi, richiede anch'esso un tempo $O(1)$.
- I metodi di accesso `root`, `left`, `right` e `parent` sono implementati direttamente in `LinkedBinaryTree` e richiedono tutti un tempo $O(1)$. I metodi `sibling`, `children` e `numChildren` sono ereditati da `AbstractBinaryTree`, dove usano un numero costante di invocazioni di altri metodi di accesso, per cui anche il loro tempo d'esecuzione è $O(1)$.
- I metodi `isInternal` e `isExternal`, ereditati dalla classe `AbstractTree`, usano un'invocazione di `numChildren`, quindi, di nuovo, vengono eseguiti in un tempo $O(1)$. Il metodo `isRoot`, pure implementato in `AbstractTree`, effettua un confronto con il risultato restituito da un'invocazione del metodo `root`, quindi viene eseguito in un tempo $O(1)$.
- Il metodo modificatore `set` viene ovviamente eseguito in un tempo $O(1)$. Più significativo è il fatto che anche tutti gli altri metodi modificatori (`addRoot`, `addLeft`, `addRight`, `attach` e `remove`) vengono eseguiti in un tempo $O(1)$: infatti, tutti effettuano, per ciascuna invocazione, un numero costante di modifiche a collegamenti di tipo genitore-figlio.
- I metodi `depth` e `height` sono stati analizzati nel Paragrafo 8.1.3. Il metodo `depth` relativo alla posizione p richiede un tempo d'esecuzione $O(d_p + 1)$, dove d_p è la profondità di p ; il metodo `height` che calcola l'altezza della radice dell'albero richiede un tempo $O(n)$.

Lo spazio di memoria complessivamente occupato da questa struttura dati è $O(n)$ per un albero avente n nodi, perché serve un esemplare della classe `Node` per ogni nodo, oltre ai campi `size` e `root` della classe esterna. La Tabella 8.1 riassume le prestazioni dell'implementazione di albero binario mediante struttura concatenata.

Tabella 8.1: Tempi d'esecuzione dei metodi di un albero binario avente n nodi implementato mediante una struttura concatenata. Lo spazio utilizzato è $O(n)$.

Metodo	Tempo d'esecuzione
<code>size</code> , <code>isEmpty</code>	$O(1)$
<code>root</code> , <code>parent</code> , <code>left</code> , <code>right</code> , <code>sibling</code> , <code>children</code> , <code>numChildren</code>	$O(1)$
<code>isInternal</code> , <code>isExternal</code> , <code>isRoot</code>	$O(1)$
<code>addRoot</code> , <code>addLeft</code> , <code>addRight</code> , <code>set</code> , <code>attach</code> , <code>remove</code>	$O(1)$
<code>depth(p)</code>	$O(d_p + 1)$
<code>height</code>	$O(n)$

8.3.2 Albero binario rappresentato con un array

Una rappresentazione alternativa per l'albero binario T è basata su un'apposita strategia di numerazione delle posizioni di T . Per ogni posizione p di T , sia $f(p)$ il numero intero definito come segue.

- Se p è la radice di T , allora $f(p) = 0$.
- Se p è il figlio sinistro della posizione q , allora $f(p) = 2f(q) + 1$.
- Se p è il figlio destro della posizione q , allora $f(p) = 2f(q) + 2$.

La funzione di numerazione f così definita è anche nota come *numerazione per livelli* (*level numbering*) delle posizioni di un albero binario T , perché numera le posizioni di ciascun livello di T in ordine crescente da sinistra a destra (come si può vedere nella Figura 8.10). Occorre osservare che la numerazione per livelli è basata sulle posizioni *potenziali* all'interno di un albero, non sull'effettiva forma di uno specifico albero, per cui i numeri assegnati ai nodi non sono necessariamente consecutivi. Ad esempio, nella Figura 8.10(b) non c'è alcun nodo a cui venga assegnato il numero 13 o 14 dalla numerazione per livelli, perché il nodo a cui è stato assegnato il numero 6 non ha figli.

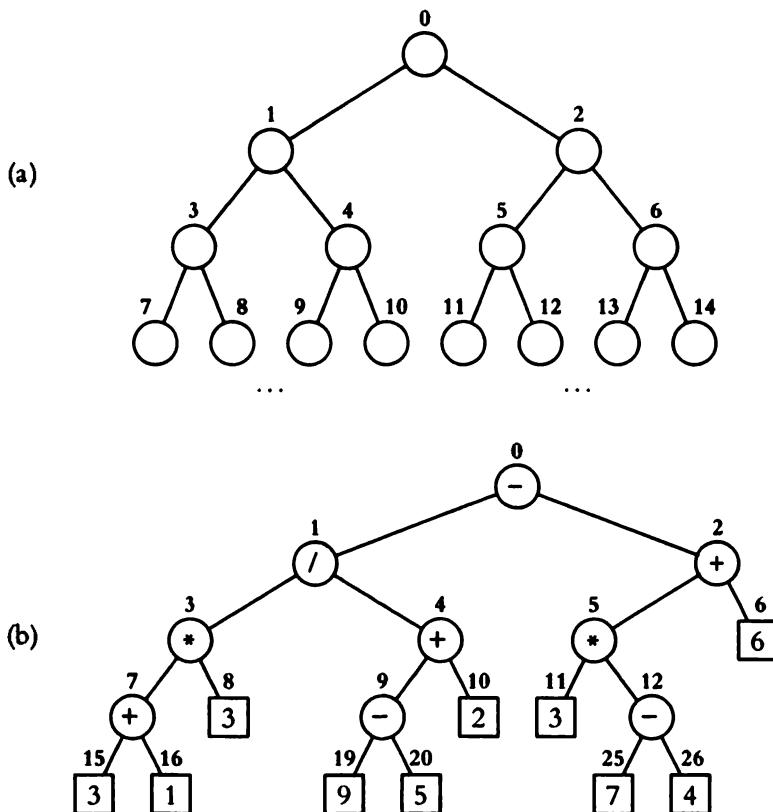


Figura 8.10: Numerazione per livelli di un albero binario: (a) schema generale; (b) un esempio.

La funzione di numerazione per livelli, f , suggerisce una possibile rappresentazione di un albero binario T mediante un array A , con l'elemento che si trova nella posizione p di T che viene memorizzato nella cella dell'array avente indice $f(p)$, come si può vedere nella Figura 8.11.

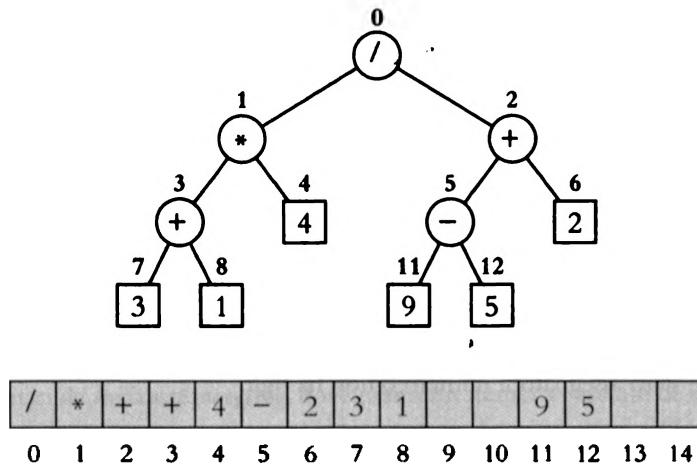


Figura 8.11: Rappresentazione di un albero binario mediante un array.

Uno dei vantaggi della rappresentazione di un albero binario mediante un array è che la posizione p può essere associata a un singolo numero intero $f(p)$ e che i metodi basati sulle posizioni, come `root`, `parent`, `left` e `right`, si possono realizzare usando semplici operazioni aritmetiche che coinvolgano il numero $f(p)$. Sulla base della nostra formula di numerazione per livelli, il figlio sinistro di p ha indice $2f(p) + 1$, il figlio destro di p ha indice $2f(p) + 2$ e il genitore di p ha indice $\lfloor(f(p) - 1)/2\rfloor$. Lasciamo all'Esercizio R-8.16 i dettagli di un'implementazione completa basata su array.

L'utilizzo dello spazio di memoria di una rappresentazione basata su array è fortemente dipendente dalla forma dell'albero. Sia n il numero di nodi di T e sia f_M il valore massimo della funzione di numerazione $f(p)$ calcolata per tutti i nodi di T . L'array A deve avere, quindi, una lunghezza pari a $N = 1 + f_M$, perché gli elementi troveranno posto nelle celle che vanno da $A[0]$ a $A[f_M]$. Si osservi che A può avere un certo numero di celle vuote, che non fanno riferimento a posizioni esistenti in T . Infatti, nel caso peggiore, $N = 2^n - 1$: la dimostrazione di questa proprietà è lasciata come esercizio (R-8.14). Nel Paragrafo 9.3 vedremo una categoria di alberi binari, chiamati *heap*, nei quali $N = n$: quindi, nonostante l'occupazione di spazio nel caso peggiore appena citata, esistono applicazioni per le quali la rappresentazione di un albero binario mediante array è efficiente in termini di spazio richiesto in memoria, anche se, nel caso di alberi binari generici, i requisiti esponenziali di tale rappresentazione sono proibitivi.

Un altro svantaggio della rappresentazione mediante array è che non riesce a fornire un supporto efficiente a molte operazioni di modifica degli alberi. Ad esempio, la rimozione di un nodo e la "promozione" del suo unico figlio richiede un tempo $O(n)$, perché non è soltanto quel figlio a dover cambiare di posto nell'array: lo devono fare anche tutti i suoi discendenti.

8.3.3 Struttura concatenata per alberi generici

Quando si rappresenta un albero binario con una struttura concatenata, ciascun nodo gestisce esplicitamente i campi `left` e `right` come riferimenti ai suoi (potenziali) figli, ma

in un albero generico non esiste un limite a priori per il numero di figli di un nodo. Una strategia naturale per realizzare un albero generico T mediante una struttura concatenata è quella di fare in modo che ciascun nodo memorizzi al proprio interno un unico *contenitore* di riferimenti ai propri figli. Ad esempio, il campo `children` di un nodo potrebbe essere un array o una lista di riferimenti ai figli del nodo (se ce ne sono): una tale rappresentazione concatenata è illustrata schematicamente nella Figura 8.12.

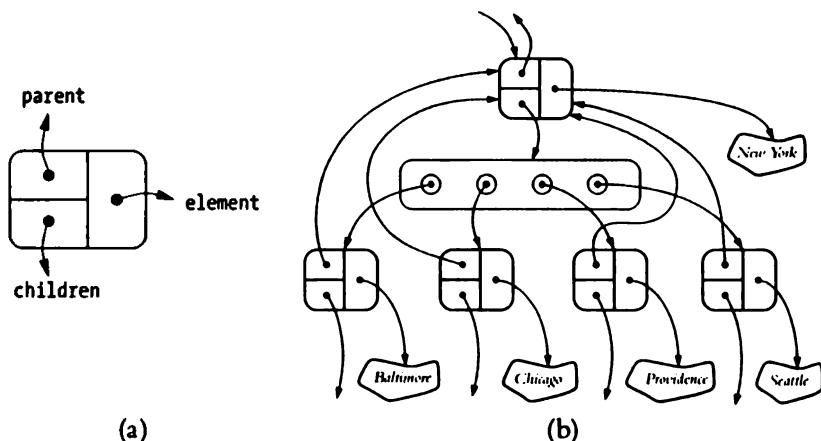


Figura 8.12: La struttura concatenata di un albero generico: (a) la struttura di un nodo; (b) una porzione della struttura dati che rappresenta un nodo e i suoi figli.

La Tabella 8.2 riassume le prestazioni dell'implementazione di un albero generico mediante una struttura concatenata. L'analisi è lasciata come esercizio (R-8.13), ma osserviamo che, usando un contenitore per memorizzare i figli di ciascuna posizione p , possiamo implementare il metodo `children(p)` effettuando semplicemente una scansione di tale contenitore.

Tabella 8.2: Tempi d'esecuzione dei metodi di accesso di un albero binario avente n nodi implementato mediante una struttura concatenata. Indichiamo con c_p il numero di figli della posizione p e con d_p la sua profondità. Lo spazio utilizzato è $O(n)$.

Metodo	Tempo d'esecuzione
<code>size</code> , <code>isEmpty</code>	$O(1)$
<code>root</code> , <code>parent</code> , <code>isRoot</code> , <code>isInternal</code> , <code>isExternal</code>	$O(1)$
<code>numChildren(p)</code>	$O(1)$
<code>children(p)</code>	$O(c_p + 1)$
<code>depth(p)</code>	$O(d_p + 1)$
<code>height</code>	$O(n)$

8.4 Algoritmi di attraversamento di alberi

Un *attraversamento* (*traversal*) di un albero T è un modo sistematico per ispezionare (o, come spesso si dice in questo contesto, "visitare") tutte le posizioni di T . L'azione specifica

associata alla “visita” di una posizione p dipende dall’applicazione che usa l’attraversamento e potrebbe essere qualunque cosa, dall’incremento di un contatore all’esecuzione di elaborazioni complesse che coinvolgano p . In questo paragrafo descriveremo alcuni schemi molto diffusi di attraversamento di alberi, implementandoli nel contesto delle nostre diverse classi che rappresentano alberi e presentando alcune loro applicazioni.

8.4.1 Attraversamenti in pre-ordine e post-ordine per alberi generici

Nell’**attraversamento in pre-ordine** o in ordine anticipato (*preorder traversal*) dell’albero T , per prima cosa si visita la radice di T , poi si attraversano ricorsivamente i sottoalberi aventi radice nei suoi figli. Se l’albero è ordinato, i sottoalberi vengono attraversati secondo l’ordine in cui sono disposti i figli. Il Codice 8.12 riporta lo pseudocodice che descrive l’algoritmo di attraversamento in pre-ordine del sottoalbero avente radice nella posizione p .

Codice 8.12: Algoritmo che esegue l’attraversamento in pre-ordine del sottoalbero avente radice nella posizione p di un albero.

Algoritmo $\text{preorder}(p)$:

```
effettua l’azione di “visita” per la posizione  $p$  { prima della ricorsione }
for ogni figlio  $c$  in  $\text{children}(p)$  do
     $\text{preorder}(c)$  { attraversa ricorsivamente il sottoalbero avente radice in  $c$  }
```

La Figura 8.13 evidenzia l’ordine in cui vengono visitate le posizioni di un albero, usato come esempio, durante l’esecuzione dell’algoritmo di attraversamento in pre-ordine.

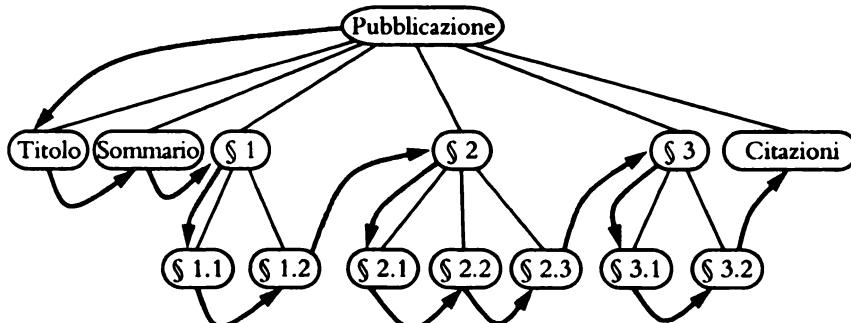


Figura 8.13: Attraversamento in pre-ordine di un albero ordinato, dove i figli di ciascuna posizione sono ordinati (come al solito) da sinistra a destra.

Attraversamento in post-ordine

Un altro importante algoritmo di attraversamento di un albero è l’**attraversamento in post-ordine** o in ordine posticipato (*postorder traversal*). In un certo senso, questo algoritmo può essere considerato l’opposto dell’attraversamento in pre-ordine, perché per prima cosa attraversa ricorsivamente i sottoalberi aventi radice nei figli della radice, poi visita la radice (da cui il nome “post-ordine”). Nel Codice 8.13 riportiamo lo pseudocodice

dell'attraversamento in post-ordine, mentre la Figura 8.14 è un esempio di tale attraversamento.

Codice 8.13: Algoritmo che esegue l'attraversamento in post-ordine del sottoalbero avente radice nella posizione p di un albero.

Algoritmo $\text{postorder}(p)$:

```
for ogni figlio  $c$  in  $\text{children}(p)$  do
     $\text{postorder}(c)$  { attraversa ricorsivamente il sottoalbero avente radice in  $c$  }
    effettua l'azione di "visita" per la posizione  $p$  { dopo la ricorsione }
```

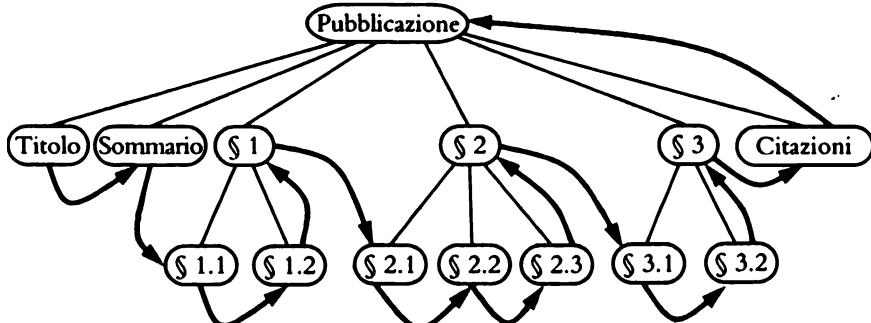


Figura 8.14: Attraversamento in post-ordine dell'albero ordinato già utilizzato nella Figura 8.13.

Analisi del tempo d'esecuzione

Gli algoritmi di attraversamento in pre-ordine e in post-ordine sono strategie efficienti per accedere a tutte le posizioni di un albero. L'analisi del tempo d'esecuzione di questi due algoritmi di attraversamento è simile a quella dell'algoritmo `height`, descritto nel Codice 8.5 e analizzato nel Paragrafo 8.1.3. Per ogni posizione p , la parte non ricorsiva dell'algoritmo di attraversamento richiede un tempo $O(c_p + 1)$, dove c_p è il numero di figli di p , nell'ipotesi che l'azione di "visita" richieda un tempo $O(1)$. In base alla Proposizione 8.4, quindi, il tempo complessivo richiesto per l'attraversamento dell'albero T è $O(n)$, dove n è il numero di posizioni di T . Tale tempo d'esecuzione è asintoticamente ottimo, dal momento che l'attraversamento deve visitare tutte le n posizioni dell'albero.

8.4.2 Attraversamento in ampiezza (*breadth-first*) di un albero

Nonostante gli attraversamenti in pre-ordine e in post-ordine siano meccanismi di visita di tutte le posizioni di un albero molto utilizzati, un altro approccio di attraversamento di un albero prevede di visitare tutte le posizioni aventi profondità d prima di visitare qualsiasi posizione avente profondità $d + 1$. Tale algoritmo prende il nome di **attraversamento in ampiezza** (*breadth-first traversal*).

Un attraversamento in ampiezza è un approccio molto diffuso nei software di gioco. Un *albero di gioco* (*game tree*) rappresenta le possibili scelte di mosse che possono essere eseguite da un giocatore (o dal calcolatore) durante una partita, con la radice dell'albero che rappresenta la configurazione iniziale del gioco. Ad esempio, la Figura 8.15 mostra una

parte dell'albero di gioco per il Tic-Tac-Toe (nomè anglosassone del gioco del *tris*, detto anche *schiera* o *filetto*).

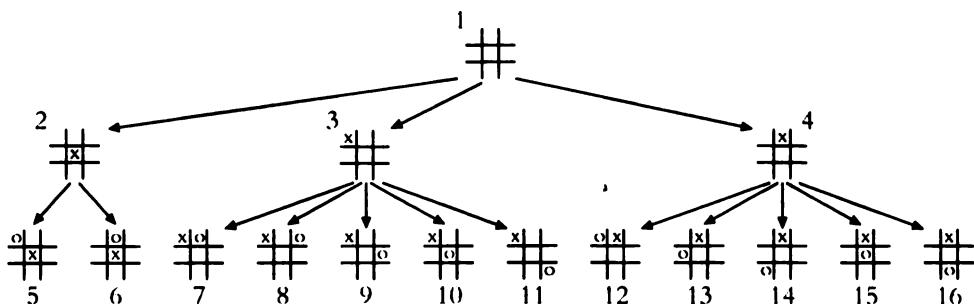


Figura 8.15: Una parte dell'albero di gioco del tris, ignorando le simmetrie. I numeri indicano l'ordine in cui le posizioni vengono visitate da un attraversamento in ampiezza.

Spesso i calcolatori attraversano in ampiezza tale albero di gioco perché non sono in grado di esplorarlo completamente in una quantità di tempo limitata. Quindi, il programma prende in esame tutte le possibili mosse, poi le risposte a tali mosse, e così via, andando tanto in profondità nell'albero quanto è consentito dal tempo a disposizione.

Il Codice 8.14 riporta lo pseudocodice per l'attraversamento in ampiezza. La procedura non è ricorsiva, perché non attraversa completamente un sottoalbero per volta. Usiamo una coda per realizzare una semantica FIFO (*first-in first-out*) per l'ordine di visita dei nodi. Il tempo d'esecuzione complessivo è ancora $O(n)$, per effetto delle n invocazioni del metodo enqueue e delle n invocazioni di dequeue.

Codice 8.14: Algoritmo che esegue l'attraversamento in ampiezza di un albero.

Algoritmo `breadthfirst()`:

```

inizializza la coda Q in modo che contenga root()
while Q non è vuota do
   $p = Q.\text{dequeue}()$  {  $p$  è il meno recente nella coda }
  effettua l'azione di "visita" per la posizione  $p$ 
  for ogni figlio  $c$  in children( $p$ ) do
    Q.enqueue( $c$ ) { aggiunge i figli di  $p$  alla coda perché siano poi visitati }
  
```

8.4.3 Attraversamento in ordine simmetrico di un albero binario

Gli attraversamenti in pre-ordine, in post-ordine e in ampiezza che abbiamo presentato per gli alberi generici possono essere applicati anche agli alberi binari. In questo paragrafo vedremo un altro diffuso algoritmo di attraversamento che, invece, è specifico per gli alberi binari.

Durante un **attraversamento in ordine simmetrico** o, semplicemente, "in ordine" (*inorder traversal*), si visita la posizione in esame dopo aver attraversato ricorsivamente il suo sottoalbero sinistro e prima di attraversare il suo sottoalbero destro. L'attraversamento in ordine

simmetrico dell'albero binario T può essere visto, informalmente, come la visita dei nodi di T "da sinistra a destra". Infatti, per ogni posizione p , l'attraversamento in ordine simmetrico visita p dopo aver visitato tutte le posizioni che si trovano nel sottoalbero sinistro di p e prima di aver visitato tutte le posizioni del sottoalbero destro di p . Nel Codice 8.15 riportiamo lo pseudocodice dell'attraversamento in ordine simmetrico, mentre la Figura 8.16 è un esempio di tale attraversamento.

Codice 8.15: Algoritmo che esegue l'attraversamento in ordine simmetrico del sottoalbero avente radice nella posizione p di un albero.

Algoritmo inorder(p):

```

if  $p$  ha il figlio sinistro  $l$  then
    inorder( $l$ ) { attraversa ricorsivamente il sottoalbero sinistro di  $p$  }
    effettua l'azione di "visita" per la posizione  $p$ 
if  $p$  ha il figlio destro  $r$  then
    inorder( $r$ ) { attraversa ricorsivamente il sottoalbero destro di  $p$  }
```

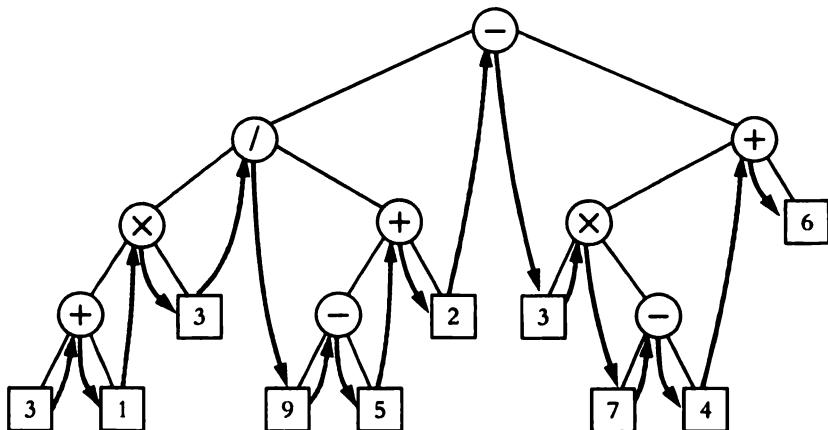


Figura 8.16: Attraversamento in ordine simmetrico di un albero binario.

L'algoritmo di attraversamento in ordine simmetrico viene usato da parecchie applicazioni interessanti. Quando si usa un albero binario per rappresentare un'espressione aritmetica, come nella Figura 8.16, l'attraversamento in ordine simmetrico visita le posizioni dell'albero in modo coerente con la rappresentazione standard delle espressioni aritmetiche, in questo caso $3 + 1 \times 3 / 9 - 5 + 2 \dots$ (anche se mancano le parentesi).

Alberi di ricerca binari

Un'applicazione molto importante dell'attraversamento in ordine simmetrico è relativa ad alberi binari che memorizzano una sequenza ordinata di elementi, secondo una struttura che chiamiamo *albero di ricerca binario* (*binary search tree*). Sia S un insieme di elementi (privi di duplicati) dotato di una relazione d'ordine. Ad esempio, S potrebbe essere un insieme di numeri interi. Un albero di ricerca binario relativo a S è un albero binario proprio, T , tale che per ogni posizione interna p di T sia vero che:

- la posizione p memorizza un elemento di S , che indichiamo con $e(p)$;
- gli elementi memorizzati nel sottoalbero sinistro di p (se ce ne sono) sono tutti minori di $e(p)$;
- gli elementi memorizzati nel sottoalbero destro di p (se ce ne sono) sono tutti maggiori di $e(p)$.

La Figura 8.17 mostra un esempio di albero di ricerca binario. Le proprietà appena enunciate garantiscono che un attraversamento in ordine simmetrico di un albero di ricerca binario ne visita gli elementi in ordine non decrescente.

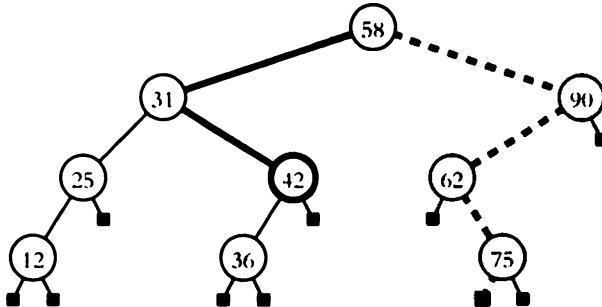


Figura 8.17: Un albero di ricerca binario che memorizza numeri interi. Il percorso evidenziato a tratto continuo viene attraversato quando si cerca (con successo) il numero 42, mentre il percorso tratteggiato è relativo alla ricerca (infruttuosa) del numero 70, che si conclude in una foglia.

Possiamo utilizzare l'albero di ricerca binario T associato all'insieme S per verificare se un determinato valore v sia presente in S , navigando lungo un percorso che scende nell'albero T a partire dalla radice. Ogni volta che incontriamo una posizione interna, p , confrontiamo il valore cercato, v , con l'elemento $e(p)$ memorizzato in p . Se $v < e(p)$, continuiamo la ricerca nel sottoalbero sinistro di p ; se $v = e(p)$, la ricerca termina con successo; se, infine, $v > e(p)$, continuiamo la ricerca nel sottoalbero destro di p . Se raggiungiamo una foglia, la ricerca termina senza successo. In altre parole, un albero di ricerca binario può essere visto come un albero binario di decisione (ricordando l'Esempio 8.5), dove in ogni nodo interno ci si chiede se l'elemento presente in quel nodo sia “minore di”, “uguale a” o “maggiore di” v , l'elemento cercato. La Figura 8.17 riporta alcuni esempi di questa operazione di ricerca.

Si osservi come il tempo d'esecuzione della ricerca in un albero di ricerca binario T sia proporzionale all'altezza di T . Ricordando la Proposizione 8.7, l'altezza di un albero binario avente n nodi può variare tra un valore minimo, $\log(n + 1) - 1$, e un valore massimo, $n - 1$. Di conseguenza, gli alberi di ricerca binari hanno la massima efficienza quando la loro altezza è minima. Il Capitolo 11 è dedicato proprio allo studio degli alberi di ricerca.

8.4.4 Implementare attraversamenti di alberi in Java

Quando abbiamo dato la prima definizione del tipo di dato astratto “albero”, nel Paragrafo 8.1.2, abbiamo scritto che l'albero T deve fornire supporto per i seguenti metodi (oltre ad altri):

iterator(): Restituisce un iteratore che scandisce tutti gli elementi.

position(): Restituisce un contenitore iterabile con tutte le posizioni dell'albero.

In quel punto della trattazione non abbiamo fatto alcuna ipotesi in merito a queste scansioni e all'ordine in cui restituiscono gli elementi e, rispettivamente, le posizioni. In questo paragrafo, invece, vedremo come qualunque algoritmo di attraversamento, tra quelli che abbiamo presentato, possa essere utilizzato per generare queste scansioni, fornendo implementazioni concrete di questi metodi da inserire nelle classi di base astratte `AbstractTree` e `AbstractBinaryTree`.

Per prima cosa, osserviamo che si può facilmente produrre una scansione di tutti gli *elementi* di un albero se si dispone di una scansione di tutte le *posizioni* di quell'albero. Il Codice 8.16 fornisce un'implementazione del metodo `iterator()` adattando opportunamente un'iterazione restituita dal metodo `positions()`: esattamente lo stesso approccio che abbiamo seguito nel Codice 7.14 del Paragrafo 7.4.2 per la classe `LinkedListPositionalList`.

Codice 8.16: Scansione di tutti gli elementi di un esemplare di una classe che deriva, direttamente o indirettamente, da `AbstractTree`, sfruttando un'analogia scansione delle posizioni dell'albero.

```

1 //----- classe ElementIterator annidata -----
2 /** Adatta l'iterazione prodotta da positions() per restituire elementi. */
3 private class ElementIterator implements Iterator<E> {
4     Iterator<Position<E>> posIterator = positions().iterator();
5     public boolean hasNext() { return posIterator.hasNext(); }
6     public E next() { return posIterator.next().getElement(); } // l'elemento!
7     public void remove() { posIterator.remove(); }
8 }
9
10 /** Restituisce un iteratore degli elementi memorizzati nell'albero. */
11 public Iterator<E> iterator() { return new ElementIterator(); }

```

Per implementare il metodo `positions()` possiamo scegliere tra diversi algoritmi di attraversamento di alberi. Dal momento che ciascuno degli ordinamenti di posizioni prodotti dai diversi algoritmi presenta alcuni vantaggi, forniamo un'implementazione pubblica di ciascuna strategia, così che possa essere scelta direttamente dall'utilizzatore della nostra classe; poi, è semplice adattare una di tali implementazioni decidendo che sia l'ordinamento predefinito che viene restituito dal metodo `positions()` della classe `AbstractTree`. Ad esempio, nel Codice 8.19 definiremo un metodo pubblico, `preorder()`, che restituisce un'iterazione delle posizioni di un albero così come viene generata da un attraversamento in pre-ordine. Come anticipato, il Codice 8.17 fa vedere come si possa definire il metodo `positions()` in modo che, banalmente, usi tale metodo `preorder()` per svolgere il proprio compito.

Codice 8.17: Definizione del metodo `positions()` in un `AbstractTree` in modo che il suo comportamento sia quello dell'algoritmo di attraversamento scelto come predefinito, l'attraversamento in pre-ordine.

```
public Iterable<Position<E>> positions() { return preorder(); }
```

Attraversamento in pre-ordine

Iniziamo dall'algoritmo di attraversamento in pre-ordine. Il nostro obiettivo è quello di scrivere nella classe `AbstractTree` un metodo pubblico, `preorder()`, che restituisca un contenitore iterabile con le posizioni dell'albero elencate in pre-ordine. Per facilitare l'implementazione, abbiamo scelto di generare un *iteratore in modalità snapshot*, così come definito nel Paragrafo 7.4.2, che restituisce una lista di tutte le posizioni (l'Esercizio C-8.47 ha come obiettivo l'implementazione di un iteratore "pigro" che, comunque, restituisca le posizioni in pre-ordine).

Per prima cosa definiamo un metodo ausiliario privato, `preorderSubtree`, presentato nel Codice 8.18, che ci consente di rendere parametrica la procedura ricorsiva sulla base di una specifica posizione dell'albero, che funge da radice del sottoalbero attraversato (passiamo come parametro anche una lista, che ha il compito di contenere le posizioni nell'ordine in cui vengono "visitate" dall'algoritmo).

Codice 8.18: Un metodo ricorsivo che esegue l'attraversamento in pre-ordine del sottoalbero avente radice `p`, una posizione di un albero. Questo codice deve essere inserito nel corpo della classe `AbstractTree`.

```

1  /** Aggiunge a snapshot le posizioni del sottoalbero avente radice p. */
2  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3      snapshot.add(p); // in pre-ordine, aggiungiamo p prima dei sottoalberi
4      for (Position<E> c : children(p))
5          preorderSubtree(c, snapshot);
6  }

```

Il metodo `preorderSubtree` ricalca l'algoritmo di alto livello originariamente descritto come pseudocodice nel Codice 8.12. Il suo caso base è implicito, perché il corpo del ciclo `for` non viene mai eseguito se la posizione ricevuta come parametro non ha figli.

Il metodo pubblico `preorder`, presentato nel Codice 8.19, ha il compito di creare una lista vuota da passare come parametro `snapshot` al metodo `preorderSubtree`, che viene invocato passando come altro parametro la radice dell'albero (nell'ipotesi che questo non sia vuoto). Come contenitore che implementi `Iterable` usiamo un esemplare di `java.util.ArrayList`.

Codice 8.19: Un metodo pubblico che effettua l'attraversamento in pre-ordine di un intero albero. Questo codice deve essere inserito nel corpo della classe `AbstractTree`.

```

1  /** Restituisce una lista delle posizioni dell'albero, in pre-ordine. */
2  public Iterable<Position<E>> preorder() {
3      List<Position<E>> snapshot = new ArrayList<>();
4      if (!isEmpty())
5          preorderSubtree(root(), snapshot); // riempie ricorsivamente snapshot
6      return snapshot;
7  }

```

Attraversamento in post-ordine

Implementiamo l'attraversamento in post-ordine in modo simile a quanto abbiamo fatto per il progetto dell'attraversamento in pre-ordine, l'unica differenza è che le posizioni "visitate" vengono aggiunte alla lista soltanto *dopo* che tutti i suoi sottoalberi sono stati traversati. Il Codice 8.20 riporta sia il metodo ausiliario ricorsivo sia il metodo pubblico.

Codice 8.20: Funzioni che consentono di effettuare l'attraversamento in post-ordine di un albero. Questo codice deve essere inserito nel corpo della classe `AbstractTree`.

```

1  /** Aggiunge a snapshot le posizioni del sottoalbero avente radice p. */
2  private void postorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3      for (Position<E> c : children(p))
4          postorderSubtree(c, snapshot);
5      snapshot.add(p); // in post-ordine, aggiungiamo p dopo i sottoalberi
6  }
7  /** Restituisce una lista delle posizioni dell'albero, in post-ordine. */
8  public Iterable<Position<E>> postorder() {
9      List<Position<E>> snapshot = new ArrayList<>();
10     if (!isEmpty())
11         postorderSubtree(root(), snapshot); // riempie ricorsivamente snapshot
12     return snapshot;
13 }
```

Attraversamento in ampiezza

Nel Codice 8.21 riportiamo l'implementazione dell'algoritmo di attraversamento in ampiezza (*breadth-first traversal*) nel contesto della classe `AbstractTree`, ricordando che si tratta di un algoritmo non ricorsivo: si basa sull'utilizzo di una coda di posizioni per gestire il processo di attraversamento. Abbiamo usato un esemplare della classe `LinkedList`, vista nel Paragrafo 6.2.3, anche se qualsiasi implementazione dell'ADT coda sarebbe adatta.

Codice 8.21: Un'implementazione dell'attraversamento in ampiezza di un albero. Questo codice deve essere inserito nel corpo della classe `AbstractTree`.

```

1  /** Restituisce una lista delle posizioni dell'albero, attraversato in ampiezza. */
2  public Iterable<Position<E>> breadthfirst() {
3      List<Position<E>> snapshot = new ArrayList<>();
4      if (!isEmpty()) {
5          Queue<Position<E>> fringe = new LinkedList<>();
6          fringe.enqueue(root()); // inizia con la radice
7          while (!fringe.isEmpty()) {
8              Position<E> p = fringe.dequeue(); // estraie dall'inizio della coda
9              snapshot.add(p); // aggiunge questa posizione
10             for (Position<E> c : children(p))
11                 fringe.enqueue(c); // aggiunge i figli in fondo alla coda
12             }
13         }
14     return snapshot;
15 }
```

Attraversamento in ordine simmetrico per alberi binari

Gli algoritmi di attraversamento in pre-ordine, in post-ordine e in ampiezza sono applicabili a tutti gli alberi, mentre l'algoritmo di attraversamento in ordine simmetrico, che si basa in modo esplicito sul concetto di figlio sinistro e figlio destro di un nodo, si applica soltanto agli alberi binari, per cui inseriamo la sua implementazione nel corpo della classe `AbstractBinaryTree`. Usiamo (nel Codice 8.22) un progetto simile a quello degli attraversamenti in pre-ordine e in post-ordine, con un metodo ausiliario ricorsivo privato per attraversare un sottoalbero.

Per molte applicazioni che usano alberi binari (si veda, ad esempio, il Capitolo 11), l'attraversamento in ordine simmetrico fornisce l'ordinamento più naturale tra le posizioni.

Per questo motivo, nel Codice 8.22 abbiamo anche fatto in modo che per la classe `AbstractBinaryTree` (e le sue sottoclassi) questo attraversamento sia quello utilizzato dal metodo `positions`, sovrascrivendo quello ereditato dalla classe `AbstractTree` (che, come abbiamo visto, usa l'attraversamento in pre-ordine). Dato che il metodo `iterator()` usa `positions()`, l'attraversamento in ordine simmetrico verrà utilizzato anche per le scansioni degli elementi di un albero binario, non solo per la scansione delle sue posizioni.

Codice 8.22: Funzioni che consentono di effettuare l'attraversamento in ordine simmetrico di un albero binario; inoltre, tale attraversamento diventa quello predefinito per le scansioni di alberi binari. Questo codice deve essere inserito nel corpo della classe `AbstractBinaryTree`.

```

1  /** Aggiunge a snapshot le posizioni del sottoalbero avente radice p. */
2  private void inorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3      if (left(p) != null)
4          inorderSubtree(left(p), snapshot);
5      snapshot.add(p);
6      if (right(p) != null)
7          inorderSubtree(right(p), snapshot);
8  }
9  /** Restituisce una lista delle posizioni dell'albero, in ordine simmetrico. */
10 public Iterable<Position<E>> inorder() {
11     List<Position<E>> snapshot = new ArrayList<>();
12     if (!isEmpty())
13         inorderSubtree(root(), snapshot); // riempie ricorsivamente snapshot
14     return snapshot;
15 }
16 /** Sovrascrive positions per rendere predefinito l'ordine simmetrico. */
17 public Iterable<Position<E>> positions() {
18     return inorder();
19 }
```

8.4.5 Applicazioni di attraversamenti di alberi

In questo paragrafo illustreremo alcune significative applicazioni di attraversamenti di alberi, tra cui anche modifiche personalizzate degli algoritmi di attraversamento standard.

Sommario di un documento

Se usiamo un albero per rappresentare la struttura gerarchica di un documento, si può utilizzare un attraversamento in pre-ordine dell'albero per generare il suo sommario (o indice). Ad esempio, il sommario associato all'albero della Figura 8.13 è visualizzato nella Figura 8.18. La parte sinistra della figura contiene semplicemente un elemento per riga, mentre la parte destra presenta gli stessi dati in modo più elegante, spostando verso destra (o, come si dice in gergo, “indentando”) ciascun elemento in base alla sua profondità nell’albero.

La versione del sommario senza rientri verso destra si può ottenere con il codice seguente, dato un albero `T` che metta a disposizione il metodo `preorder()`:

```
for (Position<E> p : T.preorder())
    System.out.println(p.getElement());
```

Pubblicazione	Pubblicazione
Titolo	Titolo
Sommario	Sommario
§1	§1
§1.1	§1.1
§1.2	§1.2
§2	§2
§2.1	§2.1
...	...
(a)	(b)

Figura 8.18: Sommario del documento rappresentato dall'albero della Figura 8.13: (a) senza rientri verso destra; (b) con rientri basati sulla profondità degli elementi all'interno dell'albero.

Per generare, invece, il sommario visibile nella Figura 8.18(b), dobbiamo far rientrare verso destra ciascun elemento per un numero di spazi uguale al doppio della profondità dell'elemento nell'albero (per cui l'elemento radice non viene fatto rientrare affatto). Nell'ipotesi che il metodo `spaces(n)` generi una stringa composta da n spazi, potremmo sostituire il corpo del ciclo precedente con l'enunciato `System.out.println(spaces(2*T.depth(p)) + p.getElement())`. Sfortunatamente, sebbene l'attraversamento in pre-ordine venga eseguito in un tempo $O(n)$, come visto nel Paragrafo 8.4.1, le invocazioni del metodo `depth` provocano un costo aggiuntivo, ancorché nascosto. Invocare `depth` per ogni posizione dell'albero richiede, come osservato analizzando l'algoritmo `heightBad` nel Paragrafo 8.1.3, un tempo che, nel caso peggiore, è $O(n^2)$.

Un approccio decisamente preferibile prevede di generare un sommario con rientri progettando in modo diverso una ricorsione che proceda, nell'albero, dall'alto verso il basso propagando la profondità della posizione in esame come parametro aggiuntivo delle invocazioni. Un'implementazione di questo tipo è presentata nel Codice 8.23 e viene eseguita in un tempo $O(n)$ nel caso peggiore (anche se, tecnicamente parlando, bisogna aggiungere il tempo speso per visualizzare le stringhe di spazi, che sono di lunghezza crescente).

Codice 8.23: Una ricorsione efficiente per visualizzare una versione con rientri degli elementi visitati dall'attraversamento in pre-ordine. Per visualizzare un intero albero `T`, la ricorsione deve essere iniziata con l'invocazione `printPreorderIndent(T, T.root(), 0)`.

```

1  /** Visualizza in pre-ordine il sottoalbero di T con radice p e profondità d. */
2  public static <E> void printPreorderIndent(Tree<E> T, Position<E> p, int d) {
3      System.out.println(spaces(2*d) + p.getElement()); // rientro basato su d
4      for (Position<E> c : T.children(p))
5          printPreorderIndent(T, c, d+1); // la profondità del figlio è d+1
6  }

```

Nell'esempio della Figura 8.18 abbiamo avuto la fortuna che la numerazione degli elementi fosse inclusa negli elementi stessi, ma, più in generale, potremmo essere interessati a usare un'attraversamento in pre-ordine per visualizzare la struttura di un albero con i rientri e aggiungendo anche una numerazione esplicita, non presente nell'albero stesso. Ad esempio, potremmo voler visualizzare l'albero della Figura 8.2 iniziando in questo modo:

```

Electronics R'Us
1 R&D
2 Sales
  2.1 Domestic
  2.2 International
    2.2.1 Canada
    2.2.2 S. America

```

Si tratta di una sfida più difficile, perché i numeri usati come etichette devono essere ricavati dalla struttura dell'albero, dove figurano sostanzialmente in forma implicita: un'etichetta dipende dal percorso che va dalla radice alla posizione corrente. Per realizzare il nostro obiettivo, aggiungiamo un ulteriore parametro alla firma del metodo ricorsivo: passeremo una lista di numeri interi che rappresentano le etichette che portano dalla radice a una particolare posizione. Ad esempio, quando staremo per visitare il nodo *Domestic*, passeremo la lista di valori {2, 1}, dalla quale si potrà dedurre la sua etichetta.

A livello di implementazione, vogliamo evitare l'inefficienza derivante dalla duplicazione di tali liste nel momento in cui dobbiamo trasmettere un nuovo parametro da un livello della ricorsione al livello successivo. Una soluzione standard consiste nel trasferire, attraverso la ricorsione, sempre lo stesso esemplare di lista: a un determinato livello della ricorsione, si aggiunge temporaneamente un nuovo elemento alla fine della lista, prima di effettuare le successive invocazioni ricorsive. Al fine di "non lasciare traccia", tale elemento aggiunto dovrà poi essere eliminato dalla lista dalla stessa invocazione ricorsiva che l'ha aggiunto. Il Codice 8.24 riporta un'implementazione basata proprio su questo approccio.

Codice 8.24: Una ricorsione efficiente per visualizzare una versione con etichette e rientri degli elementi visitati da un attraversamento in pre-ordine.

```

1  /** Visualizza con etichette il sottoalbero di T con radice p e profondità d. */
2  public static <E>
3  void printPreorderLabeled(Tree<E> T, Position<E> p, ArrayList<Integer> path) {
4      int d = path.size(); // la profondità è uguale alla lunghezza del percorso
5      System.out.println(spaces(2*d)); // visualizza il rientro, poi l'etichetta
6      for (int j=0; j < d; j++) System.out.print(path.get(j) + (j == d-1 ? " " : "."));
7      System.out.println(p.getElement());
8      path.add(1); // aggiunge al percorso il dato per il primo figlio
9      for (Position<E> c : T.children(p)) {
10          printPreorderLabeled(T, c, path);
11          path.set(d, 1 + path.get(d)); // incrementa l'ultimo dato del percorso
12      }
13      path.remove(d); // riporta il percorso al suo stato iniziale
14  }

```

Calcolo dell'occupazione di spazio su disco

Nell'Esempio 8.1 abbiamo preso in esame l'uso di un albero come modello per la struttura di un file system, con le posizioni interne e le foglie a rappresentare, rispettivamente, cartelle e file. In effetti, quando, nel Paragrafo 5.1.4, abbiamo introdotto l'uso della ricorsione, abbiamo esaminato proprio il problema del file system: anche se a quel punto non avevamo parlato di modello ad albero, era stata presentata un'implementazione di un algoritmo per calcolare l'occupazione di spazio su disco (Codice 5.5).

Il calcolo ricorsivo dello spazio occupato su disco da un file system è emblematico di un attraversamento in *post-ordine*, perché possiamo calcolare lo spazio totale occupato da

una cartella soltanto *dopo* aver calcolato lo spazio occupato dalle cartelle che ne sono figlie. Sfortunatamente l'implementazione del metodo *postorder* che abbiamo presentato nel Codice 8.20 non ha la flessibilità sufficiente a risolvere questo problema: abbiamo bisogno di un meccanismo mediante il quale i figli restituiscano informazioni al genitore durante la procedura di attraversamento dell'albero. Il Codice 8.25 mostra una soluzione *ad hoc* per il problema del calcolo dell'occupazione dello spazio su disco, dove ogni livello di ricorsione fornisce un valore all'invocante (cioè al genitore).

Codice 8.25: Calcolo ricorsivo dello spazio occupato su disco da un file system rappresentato mediante un albero. Ipotizziamo che ogni elemento dell'albero abbia come valore lo spazio occupato localmente da tale posizione.

```

1  /** Restituisce lo spazio totale su disco del sottoalbero di T avente radice p. */
2  public static int diskSpace(Tree<Integer> T, Position<Integer> p) {
3      int subtotal = p.getElement(); // per ipotesi l'elemento contiene lo spazio locale
4      for (Position<Integer> c : T.children(p))
5          subtotal += diskSpace(T, c);
6      return subtotal;
7  }

```

Rappresentazione di un albero mediante una stringa con parentesi

Conoscendo soltanto la sequenza in pre-ordine degli elementi di un albero generico, come quella di Figura 8.18a, non è possibile ricostruire la struttura dell'albero: servono informazioni aggiuntive relative al contesto dei dati. L'uso dei rientri o delle etichette numerate è un esempio di tale contesto, con una raffigurazione molto interessante per le persone umane, ma esistono rappresentazioni degli alberi mediante stringhe che sono decisamente più sintetiche e utili per i calcolatori.

In questo paragrafo vedremo una di tali rappresentazioni. La *stringa rappresentativa con parentesi* $P(T)$ dell'albero T è definita in modo ricorsivo: se T contiene una sola posizione, p , allora $P(T) = p.getElement()$; altrimenti

$$P(T) = p.getElement() + "(" + P(T_1) + ", " + \dots + ", " + P(T_k) + ")"$$

dove p è la radice di T e T_1, T_2, \dots, T_k sono i sottoalberi aventi radice nei figli di p , in ordine se T è un albero ordinato; il simbolo “+” indica qui la concatenazione tra stringhe. A titolo di esempio, la stringa rappresentativa dell'albero di Figura 8.2 è questa (andiamo a capo per esigenze editoriali):

Electronics R'Us (R&D, Sales (Domestic, International (Canada,
S. America, Overseas (Africa, Europe, Asia, Australia))),
Purchasing, Manufacturing (TV, CD, Tuner))

Anche se questa rappresentazione è sostanzialmente un attraversamento in pre-ordine, usando l'implementazione del metodo *preorder* vista in precedenza non siamo in grado di generare in modo semplice i segni di punteggiatura aggiuntivi. Le parentesi aperte devono essere aggiunte subito prima del ciclo che agisce sui figli di una posizione, le virgolette vanno aggiunte tra i figli e le parentesi chiuse devono essere inserite subito dopo il completamento del ciclo. Il metodo *parenthesize*, in Java, presentato nel Codice 8.26, è un attraversamento modificato che visualizza la stringa rappresentativa, con parentesi, di un albero T .

Codice 8.26: Metodo che visualizza la stringa rappresentativa, con parentesi, di un albero.

```

1  /** Visualizza la stringa con parentesi per il sottoalbero di T avente radice p. */
2  public static <E> void parenthesize(Tree<E> T, Position<E> p) {
3      System.out.print(p.getElement());
4      if (T.isInternal(p)) {
5          boolean firstTime = true;
6          for (Position<E> c : T.children(p)) {
7              System.out.print( (firstTime ? "(" : ", ")); // sceglie il carattere giusto
8              firstTime = false; // le prossime volte visualizzerà una virgola
9              parenthesize(T, c); // ricorsione su uno dei figli
10         }
11     System.out.print(")");
12 }
13 }
```

L'attraversamento in ordine simmetrico per disegnare alberi

L'attraversamento in ordine simmetrico può essere applicato al problema dell'elaborazione di una rappresentazione grafica di un albero binario, in analogia con quanto riportato nella Figura 8.19. Utilizziamo una convenzione molto diffusa nella grafica al calcolatore, che prevede coordinate x che aumentano procedendo da sinistra a destra e coordinate y che aumentano dall'alto verso il basso, in modo che l'origine delle coordinate stesse si trovi nell'angolo in alto a sinistra della zona di disegno.

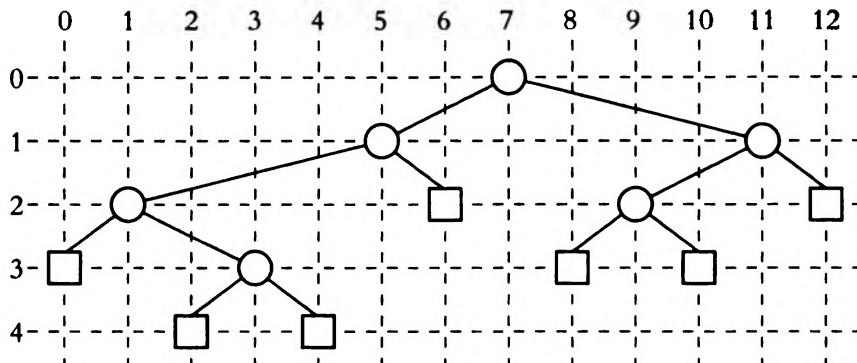


Figura 8.19: Il disegno di un albero binario mediante un attraversamento in ordine simmetrico.

Le geometria del disegno è determinata da un algoritmo che assegna le coordinate x e y a ciascuna posizione p di un albero binario T usando le regole seguenti:

- $x(p)$ è il numero di posizioni visitate prima di p nell'attraversamento in ordine simmetrico di T .
- $y(p)$ è la profondità di p in T .

Il Codice 8.27 presenta l'implementazione di un metodo ricorsivo che assegna le coordinate x e y alle posizioni di un albero. L'informazione relativa alla profondità viene passata da un livello all'altro della ricorsione, come nell'esempio precedente relativo ai rientri verso destra. Per gestire correttamente il valore della coordinata x durante l'esecuzione dell'attra-

versamento, il metodo deve disporre del valore di x che deve essere assegnato al nodo più a sinistra del sottoalbero in esame e deve restituire al proprio genitore un valore di x rivisto, in modo che sia adatto al primo nodo che verrà disegnato alla destra dello stesso sottoalbero.

Codice 8.27: Metodo ricorsivo che calcola le coordinate in cui disegnare le posizioni di un albero binario, ipotizzando che il tipo usato per gli elementi dell'albero disponga dei metodi `setX` e `setY` per impostarne le coordinate. L'invocazione iniziale del metodo deve essere `layout(T, T.root(), 0, 0)`.

```

1 public static <E> int layout(BinaryTree<E> T, Position<E> p, int d, int x) {
2     if (T.left(p) != null)
3         x = layout(T, T.left(p), d+1, x); // x verrà incrementata
4     p.getElement().setX(x++);           // post-incremento di x
5     p.getElement().setY(d);
6     if (T.right(p) != null)
7         x = layout(T, T.right(p), d+1, x); // x verrà incrementata
8     return x;
9 }
```

8.4.6 Percorso di Eulero

Le diverse applicazioni descritte nel Paragrafo 8.4.5 dimostrano la grande potenza degli attraversamenti ricorsivi di alberi, ma evidenziano anche come non tutte le applicazioni rientrino perfettamente nello schema rigido di un attraversamento in pre-ordine, in post-ordine o in ordine simmetrico. Gli algoritmi di attraversamento di un albero possono essere in qualche modo riunificati in un'unica infrastruttura di elaborazione che prende il nome di *cammino di Eulero* (*Euler tour traversal*). Il cammino di Eulero di un albero T può essere informalmente definito come un “percorso” attorno a T che inizia procedendo dalla radice verso il suo figlio più a sinistra, considerando poi i rami di T come “mura” da tenere sempre alla propria sinistra mentre si cammina (come si può vedere nella Figura 8.20).

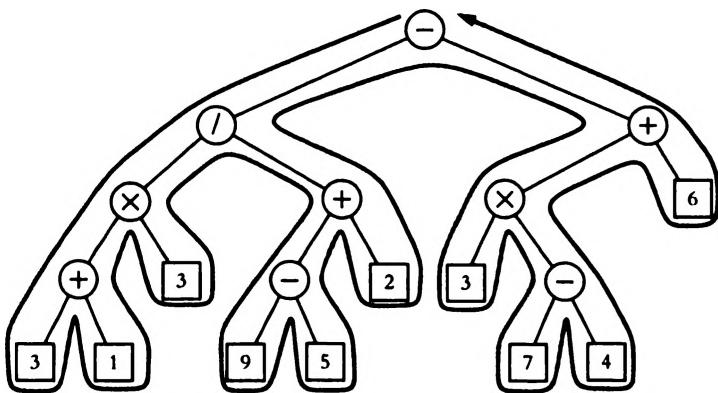


Figura 8.20: Cammino di Eulero di un albero.

La complessità del cammino è $O(n)$, per un albero avente n nodi, perché passa esattamente due volte lungo ciascuno dei rami dell'albero, il cui numero è $n - 1$: una volta scende lungo il ramo e un'altra volta risale. Per unificare i concetti di attraversamento

in pre-ordine e in post-ordine, possiamo osservare che, per ogni posizione p , sono possibili due "visite":

- Una "pre-visita" avviene quando si giunge nella posizione per la prima volta, cioè quando il cammino passa all'immediata *sinistra* del nodo, nella nostra visualizzazione
- Una "post-visita" avviene quando si giunge nella posizione per la seconda volta, risalendo verso l'alto, cioè quando il cammino passa all'immediata *destra* del nodo, nella nostra visualizzazione

La procedura che costruisce il cammino di Eulero viene vista in modo naturale come ricorsiva. Tra la pre-visita e la post-visita di una determinata posizione ci sarà un cammino costruito ricorsivamente per ciascuno dei suoi sottoalberi. Guardando, per esempio, alla Figura 8.20, vediamo che esiste una porzione ininterrotta dell'intero cammino che è essa stessa il cammino di Eulero del sottoalbero del nodo avente "/" come elemento. Tale sottocammino, a sua volta, contiene due sotto-cammini ininterrotti, uno relativo al sottoalbero sinistro di quella posizione e uno relativo al suo sottoalbero destro.

Nel caso speciale di albero binario, possiamo individuare anche il momento in cui il cammino passa immediatamente "sotto" a un nodo e decidere che tale evento sia una visita "in ordine simmetrico": avverrà subito dopo aver percorso il cammino relativo al suo sottoalbero sinistro (se questo esiste) e subito prima di percorrere il cammino relativo al suo sottoalbero destro (se esiste).

Lo pseudocodice per la costruzione del cammino di Eulero del sottoalbero avente radice nella posizione p è presentato nel Codice 8.28.

Codice 8.28: Algoritmo `eulerTour` che percorre il cammino di Eulero del sottoalbero avente radice nella posizione p di un albero.

Algoritmo `eulerTour`(T, p):

effettua l'azione di "pre-visita" per la posizione p

for ogni figlio c in $T.\text{children}(p)$ do

`eulerTour(T, c)` { ricorsivamente per ogni sottoalbero avente radice in c }

effettua l'azione di "post-visita" per la posizione p

Il cammino di Eulero estende gli attraversamenti in pre-ordine e in post-ordine, ma può anche effettuare altri tipi di attraversamenti. Supponiamo, ad esempio, di voler calcolare il numero di discendenti di ciascuna posizione p in un albero binario avente n nodi. Iniziamo un cammino di Eulero inizializzando a 0 un contatore, per poi incrementarlo di un'unità durante la pre-visita di ciascuna posizione. Per determinare il numero di discendenti di una posizione p , calcoliamo la differenza tra i valori assunti dal contatore nel momento in cui avviene la post-visita di p e il momento in cui era avvenuta la sua pre-visita, aggiungendo un'unità (per contare p , che è un discendente di se stesso per definizione). Questa regola molto semplice ci fornisce il numero di discendenti di p perché ciascun nodo appartenente al sottoalbero avente radice in p (che contiene tutti e soli i discendenti di p) viene pre-visitato (e, quindi, contato) tra il momento della pre-visita di p e il momento della post-visita di p . Quindi, abbiamo individuato un metodo che, in un tempo $O(n)$, è in grado di calcolare il numero di discendenti di ogni nodo.

Nel caso di un albero binario, possiamo modificare l'algoritmo in modo che contenga esplicitamente un'azione di "visita in ordine simmetrico" o "visita dal basso", come si può vedere nel Codice 8.29.

Codice 8.29: Algoritmo `eulerTourBinary` che percorre il cammino di Eulero del sottoalbero avente radice nella posizione p di un albero binario.

Algoritmo `eulerTourBinary`(T, p):

effettua l'azione di "pre-visita" per la posizione p

if p ha il figlio sinistro lc then

`eulerTourBinary(T, lc)` { ricorsivamente per il sottoalbero sinistro avente radice p }

effettua l'azione di "visita dal basso" per la posizione p

if p ha il figlio destro rc then

`eulerTourBinary(T, rc)` { ricorsivamente per il sottoalbero destro avente radice p }

effettua l'azione di "post-visita" per la posizione p

Ad esempio, con un cammino di Eulero siamo in grado di generare un'espressione aritmetica contenente le parentesi nel modo tradizionale, come " $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$ " per l'albero della Figura 8.20, in questo modo:

- Azione di pre-visita: se la posizione è interna, visualizza "(".
- Azione di "visita dal basso": visualizza il valore o l'operatore presente nella posizione.
- Azione di post-visita: se la posizione è interna, visualizza ")".

8.5 Esercizi

Riepilogo e approfondimento

R-8.1 Domande relative all'albero della Figura 8.3.

- Quale nodo è la radice?
- Quali sono i nodi interni?
- Quanti discendenti ha il nodo `cs016/`?
- Quanti antenati ha il nodo `cs016/`?
- Quali sono i fratelli del nodo `homeworks/`?
- Quali sono i nodi del sottoalbero avente radice nel nodo `projects/`?
- Qual è la profondità del nodo `papers/`?
- Qual è l'altezza dell'albero?

R-8.2 Definire un albero che costituisca caso peggiore per il tempo d'esecuzione dell'algoritmo `depth`.

R-8.3 Dimostrare la Proposizione 8.3.

R-8.4 Analizzando il Codice 8.5, qual è il tempo d'esecuzione di un'invocazione di `T.height(p)` quando p è una posizione diversa dalla radice dell'albero T ?

R-8.5 Descrivere un algoritmo, basato sulle operazioni definite in `BinaryTree`, che conti, in un albero binario, il numero di foglie che sono figlio *sinistro* del proprio genitore.

- R-8.6 Sia T un albero binario, avente n nodi, che può essere improprio. Descrivere come lo si possa rappresentare usando un albero binario *proprio* T' avente un numero di nodi $O(n)$.
- R-8.7 Quali sono il numero minimo e il numero massimo di nodi esterni e interni di un albero binario improprio avente n nodi?
- R-8.8 Rispondere a queste domande, che forniscono una dimostrazione della Proposizione 8.7:
- Qual è il numero minimo di nodi esterni di un albero binario proprio avente altezza h ? Giustificare la risposta fornita.
 - Qual è il numero massimo di nodi esterni di un albero binario proprio avente altezza h ? Giustificare la risposta fornita.
 - Se T è un albero binario proprio avente n nodi e altezza h , dimostrare che $\log(n+1) - 1 \leq h \leq (n-1)/2$.
 - Per quali valori di n e h i limiti inferiore e superiore per h enunciati al punto precedente possono essere effettivamente raggiunti?
- R-8.9 Dimostrare per induzione la Proposizione 8.8.
- R-8.10 Trovare il valore dell'espressione aritmetica associata a ciascun sottoalbero dell'albero binario rappresentato nella Figura 8.6.
- R-8.11 Disegnare un albero di espressione aritmetica che abbia quattro nodi esterni contenenti i numeri 1, 5, 6 e 7 (non necessariamente in questo ordine) e tre nodi interni, ciascuno dei quali contenga un operatore appartenente all'insieme $\{+, -, \times, /\}$, in modo che il valore associato alla radice sia 21. Gli operatori possono calcolare una frazione e avere frazioni come operandi, e un operatore dell'insieme può comparire più volte nell'albero.
- R-8.12 La Tabella 8.2 riporta i tempi d'esecuzione dei metodi di un albero rappresentato mediante una struttura concatenata. Dimostrare quanto riportato nella tabella fornendo, per ciascun metodo, una descrizione della sua implementazione e un'analisi del suo tempo d'esecuzione.
- R-8.13 Disegnare l'albero binario che rappresenta la seguente espressione aritmetica: “ $((5 + 2) \times (2 - 1)) / ((2 + 9) + ((7 - 2) - 1)) \times 8$ ”.
- R-8.14 Sia T un albero binario avente n nodi e sia $f(p)$ la funzione di numerazione per livelli delle posizioni di T descritta nel Paragrafo 8.3.2.
- Dimostrare che, per qualsiasi posizione p di T , $f(p) \leq 2^n - 2$.
 - Individuare un esempio di albero binario avente sette nodi che raggiunga il limite superiore di $f(p)$ enunciato al punto precedente per qualche posizione p .
- R-8.15 Mostrare come si possa usare un cammino di Eulero per calcolare il numero di livello, $f(p)$, definito nel Paragrafo 8.3.2, per ciascuna posizione di un albero binario T .
- R-8.16 Sia T un albero binario avente n posizioni, realizzato con un array A , e sia $f(p)$ la funzione di numerazione per livelli delle posizioni di T descritta nel Paragrafo 8.3.2. Fornire le descrizioni, mediante pseudocodice, di ciascuno dei metodi `root`, `parent`, `left`, `right`, `isExternal` e `isRoot`.
- R-8.17 La nostra definizione di funzione di numerazione per livelli, $f(p)$, vista nel Paragrafo 8.3.2, assegna alla radice il numero 0, ma alcuni preferiscono usare una funzione

di numerazione $g(p)$, sempre agente per livelli, che assegna alla radice il numero 1, perché questo semplifica l'aritmetica necessaria per trovare le posizioni adiacenti. Ripetere l'Esercizio R-8.16 usando la funzione di numerazione $g(p)$.

R-8.18 In quale ordine vengono visitate le posizioni dell'albero di Figura 8.6 durante un attraversamento in pre-ordine?

R-8.19 In quale ordine vengono visitate le posizioni dell'albero di Figura 8.6 durante un attraversamento in post-ordine?

R-8.20 Sia T un albero ordinato avente più di un nodo. È possibile che l'attraversamento in pre-ordine di T visiti i suoi nodi nello stesso ordine in cui vengono visitati dall'attraversamento in post-ordine di T ? Se sì, fornire un esempio; altrimenti, spiegare perché questo non può succedere. Analogamente, è possibile che l'attraversamento in pre-ordine di T visiti i suoi nodi in ordine inverso rispetto a quello in cui vengono visitati dall'attraversamento in post-ordine di T ? Se sì, fornire un esempio; altrimenti, spiegare perché questo non può succedere.

R-8.21 Rispondere alle domande dell'esercizio precedente nel caso in cui T sia un albero binario proprio avente più di un nodo.

R-8.22 Disegnare un albero binario T che soddisfi contemporaneamente le seguenti proprietà:

- ogni nodo interno di T contiene un solo carattere;
- l'attraversamento in pre-ordine di T genera la sequenza EXAMFUN;
- l'attraversamento in ordine simmetrico di T genera la sequenza MAFXUEN.

R-8.23 Riprendendo in esame l'esempio di attraversamento in ampiezza visto nella Figura 8.15 e usando i numeri di quella figura come etichette, descrivere il contenuto della coda prima di ogni passo del ciclo `while` presente nel Codice 8.14. Per iniziare, il contenuto della coda è $\{1\}$ prima del primo passo e $\{2, 3, 4\}$ prima del secondo passo.

R-8.24 Descrivere ciò che viene visualizzato dal metodo `parenthesize(T, T.root())`, presentato nel Codice 8.26, quando T è l'albero della Figura 8.6.

R-8.25 Spiegare come si può modificare il metodo `parenthesize`, visto nel Codice 8.26, in modo che usi il metodo `length()` della classe `String` per visualizzare la stringa con parentesi che rappresenta un albero andando a capo per far sì che la descrizione dell'albero possa trovar posto in una finestra di testo la cui ampiezza è 80 caratteri.

R-8.26 Qual è il tempo d'esecuzione del metodo `parenthesize(T, T.root())`, presentato nel Codice 8.26, quando T è un albero con n nodi?

Creatività

C-8.27 Descrivere un algoritmo efficiente per convertire una stringa con parentesi, correttamente composta, nell'albero equivalente. L'albero equivalente a una stringa viene definito in modo ricorsivo: la coppia di parentesi più esterne è associata alla radice e ogni sottostringa all'interno di tale coppia (definendo le sottostringhe come sequenze di caratteri comprese tra una coppia di parentesi corrispondenti) è associata a un sottoalbero della radice.

C-8.28 La *lunghezza dei percorsi* (*path length*) di un albero T è la somma delle profondità di tutte le posizioni di T . Descrivere un metodo che, in un tempo lineare in funzione del numero di posizioni dell'albero, calcoli tale somma.

C-8.29 Definiamo la *lunghezza dei percorsi interni* (*internal path length*), $I(T)$, di un albero T come la somma delle profondità di tutte le posizioni interne di T . Analogamente, Definiamo la *lunghezza dei percorsi esterni* (*external path length*), $E(T)$, di un albero T come la somma delle profondità di tutte le posizioni esterne di T . Dimostrare che, se T è un albero binario proprio avente n posizioni, è valida la relazione $E(T) = I(T) + n - 1$.

C-8.30 Sia T un albero binario (non necessariamente proprio) avente n nodi e sia D la somma delle profondità di tutti i suoi nodi esterni. Dimostrare che, se T ha il minimo numero possibile di nodi esterni, allora D è $O(n)$, mentre se T ha il massimo numero possibile di nodi esterni, allora D è $O(n \log n)$.

C-8.31 Sia T un albero binario (non necessariamente proprio) avente n nodi e sia D la somma delle profondità di tutti i suoi nodi esterni. Descrivere un albero T tale che D sia $\Omega(n^2)$. Un albero di questo tipo sarebbe il caso peggiore per il tempo d'esecuzione asintotico del metodo `heightBad`, riportato nel Codice 8.4.

C-8.32 Per un albero T , sia n_I il numero dei suoi nodi interni e sia n_E il numero dei suoi nodi esterni. Dimostrare che, se ogni nodo interno di T ha esattamente 3 figli, allora $n_E = 2n_I + 1$.

C-8.33 Due alberi ordinati, T' e T'' , sono detti *isomorfi* se è valida una delle proprietà seguenti:

- T' e T'' sono entrambi vuoti.
- T' e T'' hanno entrambi un solo nodo.
- Le radici di T' e T'' hanno lo stesso numero $k \geq 1$ di sottoalberi e l' i -esimo sottoalbero di T' è isomorfo all' i -esimo sottoalbero di T'' per $i = 1, \dots, k$.

Progettare un algoritmo che verifichi se due alberi ordinati sono isomorfi. Qual è il tempo d'esecuzione dell'algoritmo?

C-8.34 Dimostrare che esistono più di 2^n alberi binari impropri aventi n nodi interni tali che tra di essi non ci sia una coppia di alberi isomorfi, secondo la definizione data nell'Esercizio C-8.33.

C-8.35 Se si escludono gli alberi isomorfi (secondo la definizione data nell'Esercizio C-8.33), quanti alberi binari propri con esattamente 4 foglie esistono?

C-8.36 Aggiungere alla classe `LinkedBinaryTree` il metodo `pruneSubtree(p)` che elimini l'intero sottoalbero avente radice nella posizione p , garantendo che il valore della dimensione dell'albero rimanga corretto. Qual è il tempo d'esecuzione di tale metodo?

C-8.37 Aggiungere alla classe `LinkedBinaryTree` il metodo `swap(p, q)` che ristrutturi l'albero in modo che il nodo a cui fa riferimento p prenda il posto del nodo a cui fa riferimento q , e viceversa. Fare attenzione a gestire correttamente il caso in cui i nodi siano adiacenti.

C-8.38 Possiamo semplificare alcune parti della nostra implementazione della classe `LinkedBinaryTree` se utilizziamo un unico nodo sentinella, che svolga la funzione di genitore del vero nodo radice dell'albero, che diventa, invece, il figlio sinistro del nodo sentinella. Inoltre, il nodo sentinella verrà usato al posto di `null` come valore del membro `left` o `right` di un nodo che non abbia il figlio corrispondente. Nell'ipotesi che venga utilizzata questa rappresentazione, fornire una nuova implementazione dei metodi `remove` e `attach`.

- C-8.39 Descrivere come si possa clonare un esemplare di `LinkedBinaryTree` che rappresenta un albero binario proprio usando il metodo `attach`.
- C-8.40 Descrivere come si possa clonare un esemplare di `LinkedBinaryTree` che rappresenta un albero binario (non necessariamente proprio) usando i metodi `addLeft` e `addRight`.
- C-8.41 Modificare la classe `LinkedBinaryTree` in modo che implementi formalmente l'interfaccia `Cloneable`, descritta nel Paragrafo 3.6.
- C-8.42 Descrivere un algoritmo efficiente che calcoli e visualizzi, per ciascuna posizione p di un albero T , l'elemento di p seguito dall'altezza del sottoalbero avente radice p .
- C-8.43 Descrivere un algoritmo che in un tempo $O(n)$ calcoli le profondità di tutte le posizioni di un albero T avente n nodi.
- C-8.44 Il *fattore di bilanciamento* (*balance factor*) di una posizione p interna a un albero binario proprio è la differenza tra le altezze dei sottoalberi destro e sinistro di p . Dimostrare come si possa modificare il cammino di Eulero, visto nel Paragrafo 8.4.6, in modo che visualizzi i fattori di bilanciamento di tutti i nodi interni di un albero binario proprio.
- C-8.45 Progettare algoritmi per eseguire le seguenti operazioni su un albero binario T :
- `preorderNext(p)`: Restituisce la posizione visitata dopo p nell'attraversamento in pre-ordine di T (oppure `null` se p è l'ultimo nodo che viene visitato).
 - `inorderNext(p)`: Restituisce la posizione visitata dopo p nell'attraversamento in ordine simmetrico di T (oppure `null` se p è l'ultimo nodo che viene visitato).
 - `postorderNext(p)`: Restituisce la posizione visitata dopo p nell'attraversamento in post-ordine di T (oppure `null` se p è l'ultimo nodo che viene visitato).

Quali sono i tempi d'esecuzione nel caso peggiore per questi algoritmi?

- C-8.46 Descrivere, mediante pseudocodice, un metodo non ricorsivo per eseguire l'attraversamento in ordine simmetrico di un albero binario in un tempo lineare.
- C-8.47 Per implementare il metodo `preorder` nella classe `AbstractTree` abbiamo sfruttato la comodità di poter "scattare una fotografia" del contenuto dell'albero, cioè di usare un iteratore in modalità *snapshot*. Realizzare un diverso metodo `preorder` che, invece, crei un iteratore "pigro", come descritto nel Paragrafo 7.4.2.
- C-8.48 Ripetere l'Esercizio C-8.47 implementando il metodo `postorder` della classe `AbstractTree`.
- C-8.49 Ripetere l'Esercizio C-8.47 implementando il metodo `inorder` della classe `AbstractBinaryTree`.
- C-8.50 L'algoritmo `preorderDraw` disegna un albero binario T assegnando le coordinate x e y a ciascuna posizione p in modo che $x(p)$ sia il numero di nodi che precede p nell'attraversamento in pre-ordine di T e $y(p)$ sia la profondità di p in T .
- a. Dimostrare che la rappresentazione grafica di T generata da `preorderDraw` non presenta alcuna coppia di rami che si incrociano.
 - b. Ridisegnare l'albero binario della Figura 8.19 usando `preorderDraw`.
- C-8.51 Risolvere nuovamente il problema precedente per l'algoritmo `postorderDraw`, simile a `preorderDraw` tranne per il fatto che $x(p)$ diventa il numero di nodi che precede la posizione p nell'attraversamento in post-ordine.

C-8.52 Possiamo definire una *rappresentazione mediante albero binario T'* per un albero generico ordinato T in questo modo (osservare la Figura 8.21):

- Per ogni posizione p di T , esiste una posizione associata p' in T' .
- Se p è una foglia di T , allora p' in T' non ha un figlio sinistro; altrimenti il figlio sinistro di p' è q' , dove q è il primo figlio di p in T .
- Se p ha un fratello, q , immediatamente successivo nell'ordinamento in T , allora q' è il figlio destro di p' in T' ; altrimenti p' non ha figlio destro.

Data tale rappresentazione T' di un albero generico ordinato T , rispondere alle seguenti domande:

- a. L'attraversamento in pre-ordine di T' è equivalente all'attraversamento in pre-ordine di T ?
- b. L'attraversamento in post-ordine di T' è equivalente all'attraversamento in post-ordine di T ?
- c. L'attraversamento in ordine simmetrico di T' è equivalente a uno degli attraversamenti standard di T ? Se sì, quale?

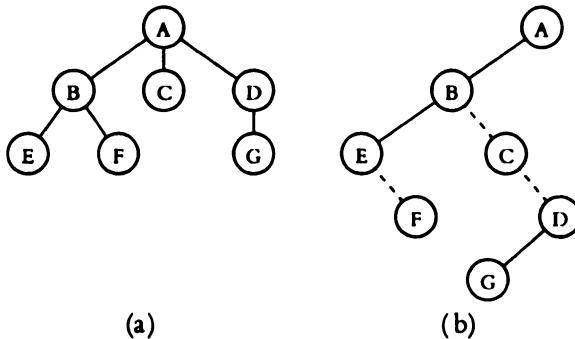


Figura 8.21: Rappresentazione di un albero mediante un albero binario: (a) un albero T ; (b) l'albero binario T' che rappresenta T . I rami tratteggiati collegano nodi di T' che sono fratelli in T .

C-8.53 Progettare un algoritmo per disegnare alberi *generici*, usando uno stile simile all'approccio usato per disegnare alberi binari mediante un attraversamento in ordine simmetrico.

C-8.54 Definiamo il *rango (rank)* di una posizione p durante un attraversamento in modo che il primo elemento visitato abbia rango 1, il secondo elemento visitato abbia rango 2, e così via. Per ogni posizione p nell'albero T , sia $\text{pre}(p)$ il rango di p nell'attraversamento in pre-ordine di T , $\text{post}(p)$ il rango di p nell'attraversamento in post-ordine di T , $\text{depth}(p)$ la profondità di p e $\text{desc}(p)$ il numero di discendenti di p , compreso p stesso. Individuare un formula che, per ogni nodo p in T , definisca $\text{post}(p)$ in funzione di $\text{desc}(p)$, $\text{depth}(p)$ e $\text{pre}(p)$.

C-8.55 Sia T un albero avente n posizioni. Definiamo il *più profondo antenato comune (LCA, lowest common ancestor)* di due posizioni, p e q , la posizioni più profonda in T che ha sia p sia q come propri discendenti (ricordando che una posizione è discendente di

se stessa). Date due posizioni p e q , descrivere un algoritmo efficiente per trovare la posizione LCA di p e q . Qual è il tempo d'esecuzione dell'algoritmo?

- C-8.56 Nell'ipotesi che ciascuna posizione p di un albero binario T sia etichettata con il proprio valore $f(p)$ della funzione di numerazione per livelli di T , progettare un metodo veloce per determinare $f(a)$ per la posizione a che sia LCA (*lowest common ancestor*, definito nell'Esercizio C-8.55) di due posizioni p e q in T , noti che siano i valori $f(p)$ e $f(q)$. Non c'è bisogno di trovare la posizione a , basta individuare il valore $f(a)$.

- C-8.57 Sia T un albero binario avente n posizioni; per ogni posizione p in T , indichiamo con d_p la profondità di p in T . La *distanza* tra due posizioni p e q in T è, per definizione, uguale a $d_p + d_q - 2d_a$, dove a è la posizione LCA di p e q (si veda l'Esercizio C-8.55 per la definizione di LCA). Il *diametro* di T è definito come la massima distanza tra due posizioni qualsiasi in T . Descrivere un algoritmo efficiente per trovare il diametro di T . Qual è il tempo d'esecuzione dell'algoritmo?

- C-8.58 Dato un albero T , la sua *rappresentazione mediante stringa con parentesi e rientri* (*indented parenthetic representation*) è una variante della rappresentazione mediante stringa con parentesi (definita nel Codice 8.26) che usa anche i rientri e le interruzioni di riga per andare a capo, come nella Figura 8.22. Progettare un algoritmo che visualizzi questa rappresentazione di albero.

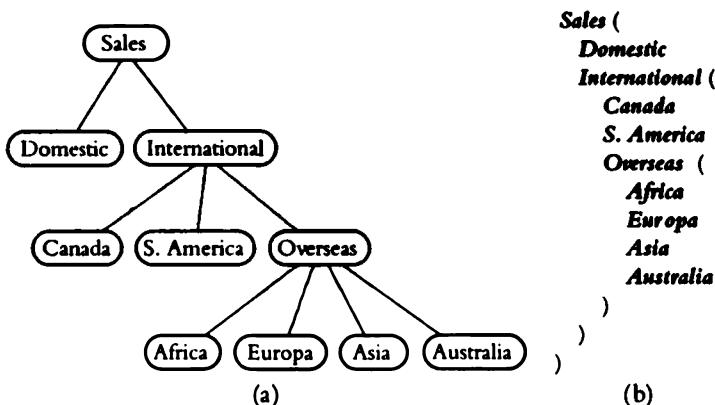


Figura 8.22: (a) Albero T ; (b) rappresentazione di T mediante stringa con parentesi e rientri.

- C-8.59 Come detto nell'Esercizio C-6.19, la *notazione postfissa* è un modo non ambiguo di scrivere un'espressione aritmetica senza fare uso di parentesi. È definita in modo che, se " $(exp_1) \text{ op } (exp_2)$ " è una normale espressione con parentesi (nella notazione aritmetica ordinaria, cioè *infissa*), dove op è un operatore, allora l'espressione equivalente in notazione postfissa è " $pexp_1 \text{ pexp}_2 \text{ op}$ ", dove $pexp_1$ è la versione postfissa di exp_1 e $pexp_2$ è la versione postfissa di exp_2 . La versione postfissa di un numero o di una variabile è proprio quel numero e quella variabile, per cui, ad esempio, la versione postfissa dell'espressione infissa " $((5 + 2) \times (8 - 3)) / 4$ " è " $5 \ 2 + 8 \ 3 - \times 4 /$ ". Descrivere un algoritmo efficiente per convertire un'espressione aritmetica in notazione infissa nella sua equivalente in notazione postfissa (suggerimento: per prima cosa, convertire l'espressione infissa nella sua rappresentazione equivalente mediante albero binario).

C-8.60 Sia T un albero binario avente n posizioni. Definiamo *posizione romana* (*Roman position*) una posizione p in T tale che il numero di discendenti di p che appartengono al suo sottoalbero sinistro differisca al massimo di 5 unità dal numero di discendenti di p che appartengono al suo sottoalbero destro. Descrivere un metodo che, in un tempo lineare, trovi tutte le posizioni p di T tali che p non sia una posizione romana, ma che tutti i discendenti di p , invece, lo siano.

Progettazione

- P-8.61 Implementare il tipo di dato astratto “albero binario” usando la rappresentazione basata su array descritta nel Paragrafo 8.3.2.
- P-8.62 Implementare il tipo di dato astratto “albero” usando una struttura concatenata, come descritto nel Paragrafo 8.3.3. Mettere a disposizione degli utilizzatori della classe un insieme ragionevole di metodi modificatori.
- P-8.63 Implementare il tipo di dato astratto “albero” usando la rappresentazione mediante albero binario descritta nell’Esercizio C-8.52, eventualmente adattando l’implementazione della classe `LinkedBinaryTree`.
- P-8.64 Lo spazio di memoria richiesto dalla classe `LinkedBinaryTree` può essere ridotto al minimo eliminando in ciascun nodo il riferimento al genitore e facendo in modo che l’implementazione di `Position` sia un oggetto che conserva al proprio interno una lista di nodi che rappresenta l’intero percorso che va dalla radice alla posizione stessa. Implementare la classe `LinkedBinaryTree` usando questa strategia.
- P-8.65 Scrivere un programma che riceva come dato in ingresso un’espressione aritmetica con parentesi e la converta in un albero di espressione binario. Il programma deve poi visualizzare l’albero in qualche modo, presentando anche il valore associato con la radice. Come ulteriore sfida, consentite alle foglie di memorizzare variabili, nella forma x_1, x_2, x_3 e così via, che valgono inizialmente zero e, poi, possono essere aggiornate in modo interattivo dal programma, su richiesta dell’utente, per poi aggiornare corrispondentemente il valore della radice visualizzato.

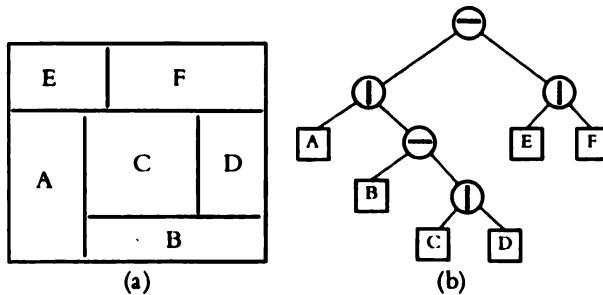


Figura 8.23: (a) Schema di pavimentazione T ; (b) albero di partizionamento associato con lo schema di pavimentazione.

- P-8.66 Uno *schema di pavimentazione* (*slicing floor plan*) divide un rettangolo (il “pavimento”), avente un lato orizzontale e un lato verticale, usando *tagli* (*cut*) orizzontali e verticali, come nella Figura 8.23a. Come si può vedere nella Figura 8.23b, uno

schema di pavimentazione può essere rappresentato da un albero binario proprio, chiamato *albero di partizionamento* (*slicing tree*), i cui nodi interni rappresentano i tagli e i cui nodi esterni rappresentano i *rettangoli elementari* (*base rectangle*) in cui il pavimento risulta decomposto dai tagli.

Definiamo ora il *problema della compattazione* (*compaction problem*) di uno schema di pavimentazione. Assumiamo che a ciascun rettangolo elementare di uno schema di pavimentazione venga assegnata una larghezza minima, w (per *width*), e un'altezza minima, h (per *height*). Il problema della compattazione consiste nel trovare le minime dimensioni (w e h) per ciascun rettangolo dello schema che sono compatibili con le dimensioni minime dei rettangoli elementari. In pratica, il problema richiede che a ciascuna posizione p dell'albero di partizionamento vengano assegnati due valori, $h(p)$ e $w(p)$, tali che:

$$w(p) = \begin{cases} w & \text{se } p \text{ è una foglia il cui rettangolo elementare} \\ & \text{ha larghezza minima } w \\ \max(w(\ell), w(r)) & \text{se } p \text{ è una posizione interna associata a un taglio} \\ & \text{orizzontale, con figlio sinistro } \ell \text{ e figlio destro } r \\ w(\ell) + w(r) & \text{se } p \text{ è una posizione interna associata a un taglio} \\ & \text{verticale, con figlio sinistro } \ell \text{ e figlio destro } r \end{cases}$$

$$h(p) = \begin{cases} & \text{se } p \text{ è una foglia il cui rettangolo elementare} \\ & \text{ha larghezza minima} \\ h(\) + h(r) & \text{se } p \text{ è una posizione interna associata a un taglio} \\ & \text{orizzontale, con figlio sinistro } \ell \text{ e figlio destro } r \\ \max(h(\), h(\)) & \text{se } p \text{ è una posizione interna associata a un taglio} \\ & \text{verticale, con figlio sinistro } \ell \text{ e figlio destro } r \end{cases}$$

Progettare una struttura dati per rappresentare schemi di pavimentazione che consenta di eseguire le seguenti operazioni:

- Creare uno schema di pavimentazione costituito da un unico rettangolo elementare.
- Decomporre un rettangolo elementare mediante un taglio orizzontale.
- Decomporre un rettangolo elementare mediante un taglio verticale.
- Assegnare a un rettangolo elementare l'altezza e la larghezza minime.
- Disegnare l'albero di partizionamento associato allo schema di pavimentazione.
- Compattare e disegnare lo schema di pavimentazione.

- P-8.67 Scrivere un programma che sia in grado di giocare in modo efficace a Tic-Tac-Toe (o *tris*, gioco spiegato nel Paragrafo 3.1.5). Per farlo, bisognerà creare un *albero di gioco T*, cioè un albero in cui ciascuna posizione corrisponde a una *configurazione di gioco*, che, in questo caso, è una rappresentazione della scacchiera del Tic-Tac-Toe (come detto nel Paragrafo 8.4.2). La radice corrisponde alla configurazione iniziale.

Per ogni posizione interna p in T , i figli di p corrispondono agli stati di gioco a cui si può giungere partendo da p ed effettuando un'unica mossa valida da parte del giocatore che è di turno, A (il primo giocatore) o B (il secondo giocatore). Le posizioni con profondità pari corrispondono alle mosse di A , mentre le posizioni con profondità dispari corrispondono alle mosse di B . Le foglie sono configurazioni finali del gioco oppure si trovano a una profondità oltre la quale non vogliamo esplorare l'albero. Assegniamo a ciascuna foglia un punteggio che indichi quanto tale configurazione sia buona per il giocatore A . Per risolvere questo problema in giochi con alberi di grandi dimensioni, come il gioco degli scacchi, dobbiamo usare una strategia euristica come funzione di assegnazione del punteggio (un'operazione che viene anche detta "valutazione"), ma per giochi semplici, come il Tic-Tac-Toe, possiamo costruire l'intero albero di gioco e assegnare alla foglie i punteggi +1, 0 o -1, indicando in questo modo, rispettivamente, la vittoria di A , un pareggio o la sconfitta di A in tale configurazione. Un algoritmo interessante per scegliere le mosse è *minimax*. Questo algoritmo assegna un punteggio a ogni posizione interna p di T , in modo che se p rappresenta un turno in cui deve giocare A , calcoliamo il punteggio di p come il valore massimo dei punteggi dei figli di p (che corrisponde alla scelta migliore per A a partire da p); se, invece, p rappresenta un turno in cui deve giocare B , calcoliamo il punteggio di p come il valore minimo dei punteggi dei figli di p (che corrisponde alla scelta migliore per B a partire da p).

- P-8.68 Scrivere un programma che riceva come dato in ingresso un albero generico T e una posizione p di T , e converta T in un altro albero avente lo stesso insieme di adiacenze tra le posizioni, ma con p come radice.
- P-8.69 Scrivere un programma che disegni un albero binario.
- P-8.70 Scrivere un programma che disegni un albero generico.
- P-8.71 Scrivere un programma che possa ricevere come dato in ingresso un albero genealogico, per poi visualizzarlo.
- P-8.72 Scrivere un programma che visualizzi il cammino di Eulero di un albero binario proprio, compresi gli spostamenti da un nodo all'altro e le azioni associate alle visite da sinistra (pre-visita), da sotto e da destra (post-visita). Illustrare il funzionamento del programma facendo in modo che calcoli e visualizzi le etichette in pre-ordine, le etichette in ordine simmetrico, le etichetta in post-ordine, il conteggio degli antenati e dei discendenti di ciascun nodo dell'albero (non necessariamente tutti contemporaneamente).

Note

Un'ampia discussione dei metodi di attraversamento degli alberi in pre-ordine, in ordine simmetrico e in post-ordine si può trovare nel libro di Knuth *Fundamental Algorithms* [60]. La tecnica del cammino di Eulero proviene dalla comunità scientifica che si occupa di algoritmi paralleli: è stata introdotta da Tarjan e Vishkin [86] e discussa da JáJá [50] e da Karp e Ramachandran [55]. L'algoritmo usato per disegnare un albero viene solitamente considerato come appartenente al *folklore* del mondo degli algoritmi che disegnano grafi. Il lettore interessato a questo argomento può consultare il libro di Di Battista, Eades, Tamassia e Tollis [29] e la panoramica scritta da Tamassia e Liotta [85]. L'Esercizio R-8.11 è stato suggerito da Micha Sharir.

9

Code prioritarie

9.1 La coda prioritaria come tipo di dato astratto

9.1.1 Priorità

Nel Capitolo 6 abbiamo introdotto la coda come tipo di dato astratto, un contenitore di oggetti che vengono inseriti e rimossi secondo la politica di gestione *FIFO* (*first-in, first-out*), cioè il primo entrato sarà il primo a uscire. Il centro telefonico di servizi alla clientela di un'azienda usa proprio questo modello, quando segnala ai clienti in attesa che “le telefonate saranno gestite secondo l'ordine in cui sono state ricevute”. In tale contesto, ogni nuova telefonata in arrivo viene inserita alla fine della coda; poi, ogni volta che un addetto al servizio clienti diventa disponibile, viene messo in contatto con il cliente la cui telefonata è appena stata rimossa dall'inizio della coda.

In realtà esistono molte applicazioni in cui viene utilizzata una struttura simile a una coda per gestire oggetti che devono essere elaborati in qualche modo, ma per le quali la politica FIFO non è sufficiente. Consideriamo, ad esempio, un centro di controllo del traffico aereo, che deve decidere quale volo debba essere autorizzato per l'atterraggio fra i tanti che si stanno avvicinando all'aeroporto. Questa scelta può essere influenzata da molti fattori, come la distanza di ciascun aereo dalla pista d'atterraggio, il tempo trascorso in modalità d'attesa o la quantità di carburante rimasto: è improbabile che le decisioni sull'ordine di atterraggio vengano prese basandosi su una rigida strategia FIFO.

Ci sono altre situazioni in cui una politica “il primo arrivato sarà il primo a essere servito” può sembrare ragionevole, anche se possono entrare in gioco altre priorità. Per usare un'altra analogia “aerea”, supponiamo che un certo volo non abbia più posti liberi a un'ora dalla partenza. A causa di possibili cancellazioni, la linea aerea gestisce una coda di passeggeri in attesa, che sperano di avere un posto. Anche se la priorità attribuita ai pas-

seggeri in attesa è influenzata dal momento in cui hanno fatto il *check-in*, sono importanti anche altri fattori, come l'importo pagato per il biglietto e il fatto di essere *frequent flyer* della compagnia, cioè, potremmo dire, "clienti affezionati". In conseguenza di tutto ciò, può accadere che un posto che si sia reso disponibile venga assegnato a un passeggero che è arrivato *più tardi* di un altro, se la compagnia aerea gli ha assegnato una priorità migliore.

In questo capitolo presentiamo un nuovo tipo di dato astratto, che chiamiamo *coda prioritaria* o *coda con priorità* (*priority queue*). Si tratta di un contenitore di elementi, ciascuno dei quali ha un livello di priorità: si può inserire qualsiasi elemento, ma è consentita la rimozione del solo elemento avente la priorità migliore. Quando un elemento viene aggiunto a una coda prioritaria, è l'utente della coda stessa a decidere quale sia la priorità di quell'elemento, fornendo la *chiave* (*key*) da associare all'elemento stesso all'interno della coda. L'elemento avente la chiave *minima* sarà, per convenzione, il prossimo a essere rimosso dalla coda (per cui un elemento avente chiave 1 avrà la priorità rispetto a un elemento avente chiave 2). Sebbene sia piuttosto frequente che le priorità siano espresse da numeri, in realtà come chiave può essere utilizzato qualunque oggetto, in Java, a patto che esista un modo per confrontare qualunque coppia di esemplari di chiave, a e b; come anche si dice, deve esistere un ordinamento totale tra le chiavi. Con una così ampia generalità, le applicazioni possono sviluppare il proprio concetto di priorità per gli elementi. Ad esempio, ciascun diverso analista finanziario può assegnare valutazioni diverse (cioè diverse priorità) a un particolare investimento, come un pacchetto azionario.

9.1.2 La coda prioritaria come ADT

Come modello di un elemento e della sua priorità usiamo un oggetto composito chiave-valore che chiamiamo genericamente *voce* (*entry*), anche se rimandiamo al Paragrafo 9.2.1 la definizione tecnica del tipo di dato *Entry*.

Definiamo il tipo di dato astratto "coda prioritaria" in modo che fornisca supporto ai metodi seguenti:

- insert(*k*, *v*):** Crea e inserisce nella coda prioritaria una voce con chiave *k* e valore *v*.
- min():** Restituisce una delle voci (*k*, *v*) presenti nella coda prioritaria che hanno chiave minima, senza eliminarla; restituisce *null* se la coda prioritaria è vuota.
- removeMin():** Elimina e restituisce una delle voci (*k*, *v*) presenti nella coda prioritaria che hanno chiave minima; restituisce *null* se la coda prioritaria è vuota.
- size():** Restituisce il numero di voci presenti nella coda prioritaria.
- isEmpty():** Restituisce *true* se e solo se la coda prioritaria è vuota.

Una coda prioritaria può contenere più voci aventi chiavi equivalenti, nel qual caso i metodi *min* e *removeMin* possono scegliere arbitrariamente tra le voci aventi chiave minima. I valori associati alle chiavi possono essere di qualsiasi tipo.

In questo nostro primo modello di coda prioritaria ipotizziamo che la chiave di un elemento rimanga fissata una volta che questo sia stato inserito nella coda prioritaria, ma

nel Paragrafo 9.5 vedremo un'estensione di questo ADT che consente all'utilizzatore di modificare la chiave di un elemento presente all'interno della coda prioritaria.

Esempio 9.1: La tabella seguente mostra una sequenza di operazioni e i loro effetti su una coda prioritaria inizialmente vuota. La colonna più a destra, che mostra il contenuto della coda prioritaria, è in qualche modo ingannevole, perché mostra le entità ordinate per chiave: una tale rappresentazione interna dei dati non è un requisito necessario per il funzionamento della coda prioritaria.

Metodo	Valore restituito	Contenuto della coda prioritaria
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

9.2 Implementare una coda prioritaria

In questo paragrafo vedremo diversi aspetti tecnici riguardanti l'implementazione di code prioritarie in Java e definiremo una classe di base astratta che fornisce alcune funzionalità che sono condivise da tutte le implementazioni di coda prioritaria descritte in questo capitolo, dopodiché forniremo due implementazioni concrete di coda prioritaria, usando una lista posizionale L come spazio di memorizzazione (si veda il Paragrafo 7.3). Queste due implementazioni differiscono tra loro per il fatto di tenere le voci ordinate in base alle chiavi oppure no.

9.2.1 L'oggetto composito Entry

Una delle difficoltà nell'implementazione di una coda prioritaria consiste nel dover tenere traccia unitariamente di elementi associati a chiavi, anche quando queste voci vengono spostate all'interno della struttura dati. Questo ci fa tornare alla mente un caso di studio, visto nel Paragrafo 7.7, dove gestivamo una lista di elementi associati a frequenze di accesso. In quella situazione avevamo introdotto lo *schema progettuale della composizione* (*composition design pattern*), definendo una classe `Item` che accoppiava, nella nostra struttura dati principale, ciascun elemento con il conteggio a esso associato. Nelle code prioritarie usiamo la composizione per accoppiare una chiave k e un valore v all'interno di un unico oggetto. Per formalizzare questo concetto, nel Codice 9.1 definiamo l'interfaccia pubblica `Entry`.

Codice 9.1: Interfaccia Java per una voce che memorizza una coppia chiave-valore.

```

1  /** Interfaccia per una coppia chiave-valore. */
2  public interface Entry<K,V> {
3      K getKey(); // restituisce la chiave memorizzata in questa voce
4      V getValue(); // restituisce il valore memorizzato in questa voce
5  }

```

Nella definizione dell'interfaccia che descrive una coda prioritaria, nel Codice 9.2, usiamo proprio il tipo `Entry`: questo ci consente di scrivere metodi, come `min` e `removeMin`, che restituiscono sia una chiave sia un valore, composti in un'unico oggetto. L'interfaccia dichiara anche il metodo `insert`, che restituisce la voce che è stata inserita nel contenitore: nella più avanzata *coda prioritaria modificabile* (*adaptable priority queue*, nel Paragrafo 9.5) tale voce può essere anche modificata o eliminata.

Codice 9.2: Interfaccia Java per il tipo di dato astratto "coda prioritaria".

```

1  /** Interfaccia per l'ADT coda prioritaria. */
2  public interface PriorityQueue<K,V> {
3      int size();
4      boolean isEmpty();
5      Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
6      Entry<K,V> min();
7      Entry<K,V> removeMin();
8  }

```

9.2.2 Confrontare chiavi totalmente ordinate

Nella definizione dell'ADT coda prioritaria, si può consentire a qualunque tipo di oggetto di svolgere il ruolo di chiave, ma deve essere possibile effettuare confronti tra chiavi e i risultati dei confronti non devono essere contraddittori. Perché la regola di confronto, che indichiamo con il simbolo \leq , sia coerente, deve definire una *relazione d'ordine totale*, cioè deve soddisfare le seguenti proprietà per qualsiasi chiave k_1 , k_2 e k_3 :

- Proprietà di confrontabilità o di totalità: $k_1 \leq k_2$ oppure $k_2 \leq k_1$.
- Proprietà antisimmetrica: se $k_1 \leq k_2$ e $k_2 \leq k_1$, allora $k_1 = k_2$.
- Proprietà transitiva: se $k_1 \leq k_2$ e $k_2 \leq k_3$, allora $k_1 \leq k_3$.

La proprietà di confrontabilità afferma che la regola di confronto è definita per qualunque coppia di chiavi e implica la seguente:

- Proprietà riflessiva: $k \leq k$.

Una regola di confronto, \leq , che definisca una relazione d'ordine totale non darà mai luogo a contraddizioni e induce un ordinamento lineare tra le chiavi appartenenti a un insieme, per cui, se un insieme (finito) di elementi è dotato di un ordinamento totale definito al proprio interno, allora il concetto di chiave *minima*, k_{\min} , è ben definito: è quella chiave che rende valida la proprietà $k_{\min} \leq k$ per ogni altra chiave k appartenente all'insieme.

L'interfaccia Comparable

Java mette a disposizione due modi diversi per definire confronti tra oggetti. Il primo di questi prevede che una classe possa definire quello che si chiama *ordinamento naturale* dei propri esemplari implementando formalmente l'interfaccia `java.lang.Comparable`, che dichiara un unico metodo: `compareTo`. L'invocazione `a.compareTo(b)` deve restituire un numero intero i con il seguente significato:

- $i < 0$ segnala che $a < b$
- $i = 0$ segnala che $a = b$
- $i > 0$ segnala che $a > b$

Ad esempio, il metodo `compareTo` della classe `String` definisce che l'ordinamento naturale delle stringhe è quello *lessicografico*, che è un'estensione a Unicode dell'ordinamento alfabetico (sensibile alle differenze tra maiuscole e minuscole).

L'interfaccia Comparator

In alcune applicazioni si vogliono confrontare oggetti sulla base di un criterio diverso da quello previsto dal loro ordinamento naturale. Ad esempio, potremmo essere interessati a sapere quale, fra due stringhe, è la più corta, oppure potremmo definire regole complesse e personali per decidere quali azioni di borsa siano più promettenti. Per fornire il supporto più generale possibile, Java definisce l'interfaccia `java.util.Comparator`. Un *comparatore* è un oggetto esterno alla classe di cui sono esemplari le chiavi che confronta e mette a disposizione un metodo, avente la firma `compare(a, b)`, che restituisce un numero intero con significato analogo a quello del metodo `compareTo` appena descritto.

Come esempio concreto, vediamo, nel Codice 9.3, la definizione di un comparatore che valuta stringhe sulla base della loro lunghezza (invece di seguire l'ordinamento naturale, cioè lessicografico).

Codice 9.3: Un comparatore che valuta stringhe in base alla loro lunghezza.

```

1  public class StringLengthComparator implements Comparator<String> {
2      /** Confronta due stringhe in base alla loro lunghezza. */
3      public int compare(String a, String b) {
4          if (a.length() < b.length()) return -1;
5          else if (a.length() == b.length()) return 0;
6          else return 1;
7      }
8  }
```

Comparatori e l'ADT coda prioritaria

Per definire una coda prioritaria generica e riutilizzabile, consentiamo all'utilizzatore di scegliere chiavi di qualsiasi tipo, fornendo al costruttore della coda prioritaria, come parametro, un esemplare di comparatore adeguato. La coda prioritaria userà quel comparatore ogni volta che dovrà confrontare tra loro due chiavi.

Per comodità, consentiamo anche la creazione di esemplari di coda prioritaria che usino l'ordinamento naturale per le chiavi fornite (nell'ipotesi che tali chiavi siano esemplari di una classe che implementa `java.lang.Comparable`). In tal caso viene costruito un esemplare della classe `DefaultComparator`, definita nel Codice 9.4.

Codice 9.4: La classe `DefaultComparator` che realizza un comparatore basato sull'ordinamento naturale del tipo di dato usato come parametro effettivo.

```

1  public class DefaultComparator<E> implements Comparator<E> {
2      public int compare(E a, E b) throws ClassCastException {
3          return ((Comparable<E>) a).compareTo(b);
4      }
5  }
```

9.2.3 La classe di base `AbstractPriorityQueue`

Per gestire le funzionalità e i problemi comuni a tutte le nostre implementazioni di coda prioritaria, nel Codice 9.5 definiamo una classe di base astratta, `AbstractPriorityQueue` (si può rivedere il Paragrafo 2.3.3 per una discussione sulle classi astratte), contenente anche una classe annidata, `PQEntry`, che implementa l'interfaccia pubblica `Entry`.

La nostra classe astratta dichiara e inizializza anche una variabile di esemplare, `comp`, che memorizza il comparatore usato dalla coda prioritaria. Definiamo, inoltre, un metodo `protected`, `compare`, che invoca il comparatore sulle chiavi delle due entità che gli vengono fornite.

Codice 9.5: La classe `AbstractPriorityQueue`. Fornisce il supporto per la gestione di un comparatore e definisce una classe annidata, `PQEntry`, che compone una chiave e un valore all'interno di un unico oggetto. Per comodità, è anche presente un'implementazione del metodo `isEmpty` basata sulla presunta presenza del metodo `size`.

```

1  /** Classe astratta che aiuta l'implementazione dell'interfaccia PriorityQueue. */
2  public abstract class AbstractPriorityQueue<K,V>
3      implements PriorityQueue<K,V> {
4      //----- classe PQEntry annidata -----
5      protected static class PQEntry<K,V> implements Entry<K,V> {
6          private K k; // chiave
7          private V v; // valore
8          public PQEntry(K key, V value) {
9              k = key;
10             v = value;
11         }
12         // metodi dell'interfaccia Entry
13         public K getKey() { return k; }
14         public V getValue() { return v; }
15         // metodi ausiliari che non fanno parte dell'interfaccia Entry
16         protected void setKey(K key) { k = key; }
17         protected void setValue(V value) { v = value; }
18     } //---- fine della classe PQEntry annidata -----
19
20     // variabile di esemplare della classe AbstractPriorityQueue
21     /** Il comparatore che definisce l'ordine tra le chiavi della coda prioritaria. */
22     private Comparator<K> comp;
23     /** Crea una coda prioritaria vuota che usa il comparatore fornito. */
24     protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }
25     /** Crea una coda prioritaria vuota che usa l'ordine naturale tra le chiavi. */
26     protected AbstractPriorityQueue() { this(new DefaultComparator<K>()); }
27     /** Metodo per confrontare due entità in base alle loro chiavi. */
28     protected int compare(Entry<K,V> a, Entry<K,V> b) {
```

```

29     return comp.compare(a.getKey(), b.getKey());
30 }
31 /**
32  * Determina se la chiave è valida.
33  */
34 protected boolean checkKey(K key) throws IllegalArgumentException {
35     try {                                     // verifica se la chiave è valida
36         return (comp.compare(key, key) == 0); // confrontandola con se stessa
37     } catch (ClassCastException e) {
38         throw new IllegalArgumentException("Incompatible key");
39     }
40 /**
41  * Verifica se la coda prioritaria è vuota.
42  */
43 public boolean isEmpty() { return size() == 0; }
44 }
```

9.2.4 Realizzare una coda prioritaria con una lista non ordinata

Nella nostra prima implementazione concreta di coda prioritaria memorizziamo le sue voci all'interno di una lista concatenata *non ordinata (unsorted)*: il Codice 9.6 presenta la nostra classe `UnsortedPriorityQueue` come sottoclasse di `AbstractPriorityQueue` (definita nel Codice 9.5). Ai fini della memorizzazione interna, le coppie chiave-valore sono rappresentate da oggetti composti, usando esemplari della classe `PQEntry` ereditata. Tali voci vengono memorizzate all'interno di un esemplare di `PositionalList` che si trova in una variabile di esemplare della coda prioritaria. Ipotizziamo che la lista posizionale sia implementata con una lista doppiamente concatenata, come nel Paragrafo 7.3, per cui tutte le operazioni di quel contenitore vengono eseguite in un tempo $O(1)$.

Cominciamo con una lista vuota: nel momento in cui la coda prioritaria viene costruita, e in ogni istante, la dimensione della lista uguaglia il numero di coppie chiave-valore presenti nella coda prioritaria. Per questo motivo il metodo `size` della coda prioritaria restituisce semplicemente la lunghezza della lista interna. Come conseguenza delle scelte di progetto effettuate per la classe `AbstractPriorityQueue`, ereditiamo un'implementazione concreta del metodo `isEmpty` che si basa proprio su un'invocazione del metodo `size` che abbiamo così definito.

Ogni volta che una coppia chiave-valore viene aggiunta alla coda prioritaria, tramite il metodo `insert`, dobbiamo individuare la voce avente la chiave minima. Dato che le voci non sono ordinate all'interno della lista, per poter trovare una delle voci aventi la chiave minima dobbiamo ispezionarle tutte. Per comodità, definiamo un metodo ausiliario privato, `findMin`, che restituisce la *posizione* di una delle voci aventi la chiave minima. La conoscenza della posizione consente al metodo `removeMin` di invocare il metodo `remove` della lista posizionale, mentre il metodo `min` usa semplicemente la posizione per recuperare la voce che serve a predisporre la coppia chiave-valore da restituire. A causa del ciclo usato per trovare la chiave minima, entrambi i metodi, `min` e `removeMin`, vengono eseguiti in un tempo $O(n)$, dove n è il numero di voci presenti nella coda prioritaria.

La Tabella 9.1 riassume i tempi d'esecuzione della coda prioritaria implementata dalla classe `UnsortedPriorityQueue`.

Tabella 9.1: Tempi d'esecuzione nel caso peggiore dei metodi di una coda prioritaria di dimensione n implementata mediante una lista doppiamente concatenata e non ordinata. Lo spazio utilizzato è $O(n)$.

Metodo	Tempo d'esecuzione
<code>size</code>	$O(1)$
<code>isEmpty</code>	$O(1)$
<code>insert</code>	$O(1)$
<code>min</code>	$O(n)$
<code>removeMin</code>	$O(n)$

Codice 9.6: Un'implementazione di coda prioritaria che usa una lista non ordinata.

La superclasse `AbstractPriorityQueue` è descritta nel Codice 9.5, mentre la classe `UnsortedPriorityQueue` è stata presentata nel Codice 7.3.

```

1  /** Un'implementazione di coda prioritaria mediante una lista non ordinata. */
2  public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3      /** Contenitore principale delle voci della coda prioritaria */
4      private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();
5
6      /** Crea una coda prioritaria vuota che usa l'ordine naturale tra le chiavi. */
7      public UnsortedPriorityQueue() { super(); }
8      /** Crea una coda prioritaria vuota che usa il comparatore fornito. */
9      public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }
10
11     /** Restituisce la Position di una delle voci aventi chiave minima. */
12     private Position<Entry<K,V>> findMin() { // invocata solo se la coda non è vuota
13         Position<Entry<K,V>> small = list.first();
14         for (Position<Entry<K,V>> walk : list.positions())
15             if (compare(walk.getElement(), small.getElement()) < 0)
16                 small = walk; // trovata una chiave ancora minore
17         return small;
18     }
19
20     /** Inserisce una coppia chiave-valore e restituisce la voce creata. */
21     public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
22         checkKey(key); // metodo ausiliario di verifica (può lanciare eccezione)
23         Entry<K,V> newest = new PQEntry<>(key, value);
24         list.addLast(newest);
25         return newest;
26     }
27
28     /** Restituisce una delle voci aventi chiave minima (senza rimuoverla). */
29     public Entry<K,V> min() {
30         if (list.isEmpty()) return null;
31         return findMin().getElement();
32     }
33
34     /** Elimina e restituisce una delle voci aventi chiave minima. */
35     public Entry<K,V> removeMin() {
36         if (list.isEmpty()) return null;
37         return list.remove(findMin());
38     }

```

```

39
40    /** Restituisce il numero di voci presenti nella coda prioritaria. */
41    public int size() { return list.size(); }
42 }
```

9.2.5 Realizzare una coda prioritaria con una lista ordinata

La nostra prossima implementazione di coda prioritaria usa ancora una lista posizionale, ma vi tiene le voci ordinate (*sotese*) in ordine di chiave non decrescente. In questo modo siamo certi che il primo elemento della lista è una voce avente la chiave minima.

Il Codice 9.7 presenta la nostra classe `SortedPriorityQueue`. L'implementazione dei metodi `min` e `removeMin` è abbastanza banale, dato che sappiamo che il primo elemento della lista ha la chiave minima: usiamo il metodo `first` della lista posizionale per trovare la posizione della prima voce e il metodo `remove` per eliminarla dalla lista. Nell'ipotesi che la lista sia implementata mediante una lista doppiamente concatenata, le operazioni `min` e `removeMin` richiedono un tempo $O(1)$.

Questo vantaggio ha, però, un costo: ora il metodo `insert` deve scandire la lista per trovare la posizione corretta in cui inserire la nuova voce. La nostra implementazione parte dalla fine della lista, spostandosi all'indietro finché la nuova chiave è minore di quella della voce in esame; nel caso peggiore, procede fino a raggiungere l'inizio della lista, per cui il metodo `insert` viene eseguito, nel caso peggiore, in un tempo $O(n)$, essendo n il numero di voci presenti nella coda prioritaria nel momento in cui il metodo viene eseguito. Riassumendo, quando si usa una lista ordinata per implementare una coda prioritaria, gli inserimenti vengono eseguiti in un tempo lineare, mentre la ricerca e la rimozione della voce con chiave minima avvengono in un tempo costante.

Confronto tra le due implementazioni basate su lista

La Tabella 9.2 confronta i tempi d'esecuzione dei metodi della coda prioritaria realizzata mediante una lista ordinata o non ordinata. Risulta evidente un interessante compromesso: una lista non ordinata consente inserimenti rapidi ma ricerche e rimozioni lente, mentre, al contrario, una lista ordinata consente ricerche e rimozioni veloci, ma inserimenti lenti.

Tabella 9.2: Tempi d'esecuzione nel caso peggiore dei metodi di una coda prioritaria di dimensione n implementata mediante una lista ordinata o non ordinata, ipotizzando che, in entrambi i casi, la lista sia stata implementata con una lista doppiamente concatenata. Lo spazio utilizzato è $O(n)$.

Metodo	Lista non ordinata	Lista ordinata
<code>size</code>	$O(1)$	$O(1)$
<code>isEmpty</code>	$O(1)$	$O(1)$
<code>insert</code>	$O(1)$	$O(n)$
<code>min</code>	$O(n)$	$O(1)$
<code>removeMin</code>	$O(n)$	$O(1)$

Codice 9.7: Un'implementazione di coda prioritaria che usa una lista ordinata. La superclasse `AbstractPriorityQueue` è descritta nel Codice 9.5, mentre la classe `LinkedPositionalList` è stata presentata nel Codice 7.3.

```

1  /** Un'implementazione di coda prioritaria mediante una lista ordinata. */
2  public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3      /** Contenitore principale delle voci della coda prioritaria */
4      private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();
5
6      /** Crea una coda prioritaria vuota che usa l'ordine naturale tra le chiavi. */
7      public SortedPriorityQueue() { super(); }
8      /** Crea una coda prioritaria vuota che usa il comparatore fornito. */
9      public SortedPriorityQueue(Comparator<K> comp) { super(comp); }
10
11     /** Inserisce una coppia chiave-valore e restituisce la voce creata. */
12     public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
13         checkKey(key); // metodo ausiliario di verifica (può lanciare eccezione)
14         Entry<K,V> newest = new PQEntry<>(key, value);
15         Position<Entry<K,V>> walk = list.last();
16         // procede a ritroso, cercando chiavi minori
17         while (walk != null && compare(newest, walk.getElement()) < 0)
18             walk = list.before(walk);
19         if (walk == null)
20             list.addFirst(newest); // la nuova chiave è quella minima
21         else
22             list.addAfter(walk, newest); // newest viene posta dopo walk
23         return newest;
24     }
25
26     /** Restituisce una delle voci aventi chiave minima (senza rimuoverla). */
27     public Entry<K,V> min() {
28         if (list.isEmpty()) return null;
29         return list.first().getElement();
30     }
31
32     /** Elimina e restituisce una delle voci aventi chiave minima. */
33     public Entry<K,V> removeMin() {
34         if (list.isEmpty()) return null;
35         return list.remove(list.first());
36     }
37
38     /** Restituisce il numero di voci presenti nella coda prioritaria. */
39     public int size() { return list.size(); }
40 }
```

9.3 Heap

Le due strategie di implementazione dell'ADT coda prioritaria presentate nel paragrafo precedente sono un compromesso interessante. Memorizzando le voci in una lista *non ordinata*, possiamo eseguire inserimenti in un tempo $O(1)$, ma per cercare o eliminare uno degli elementi aventi chiave minima serve un ciclo che scandisca l'intero contenitore, in un tempo $O(n)$. Al contrario, usando una lista *ordinata*, possiamo banalmente cercare o rimuovere l'elemento minimo in un tempo $O(1)$, ma l'aggiunta di un nuovo elemento alla coda può richiedere un tempo $O(n)$ per il ripristino dell'ordinamento.

In questo paragrafo descriviamo una realizzazione di coda prioritaria più efficiente, usando una struttura dati chiamata *heap binario* o, semplicemente, *heap* (che significa letteralmente “mucchio”). Questa struttura dati ci consentirà di effettuare tanto gli inserimenti quanto le rimozioni in un tempo logaritmico, che costituisce un miglioramento davvero significativo rispetto alle implementazioni basate su lista che abbiamo analizzato nel Paragrafo 9.2. Il meccanismo fondamentale che consente a uno *heap* di ottenere questo miglioramento consiste nell'utilizzo di un albero binario, per trovare un compromesso tra il fatto di avere gli elementi completamente ordinati o, viceversa, non avere nessuna proprietà di ordinamento tra loro.

9.3.1 La struttura dati *heap*

Come si può vedere nella Figura 9.1, uno *heap* è un albero binario T che memorizza voci nelle proprie posizioni e che soddisfa due ulteriori proprietà: una proprietà relazionale definita in termini delle modalità di memorizzazione delle chiavi in T e una proprietà strutturale che riguarda la forma stessa di T . La proprietà relazionale è la seguente:

Proprietà di ordinamento di uno *heap*: In uno *heap* T , per ogni posizione p diversa dalla radice, la chiave memorizzata in p è non minore della chiave memorizzata nel genitore di p .

In conseguenza della proprietà di ordinamento di uno *heap*, le chiavi che si trovano lungo un percorso che parte dalla radice e arriva a una qualsiasi foglia di T sono in ordine non decrescente. Inoltre, la chiave minima è sempre memorizzata nella radice di T : questo rende estremamente semplice la localizzazione di una delle voci aventi la chiave minima nel momento in cui viene invocato il metodo `min` o il metodo `removeMin`, perché questa si trova “in cima al mucchio” (dà cui il nome “mucchio”, *heap*, dato alla struttura). È bene segnalare che la struttura dati *heap* definita in questo contesto non ha nulla a che vedere con la “memoria *heap*” usata nell’ambiente di esecuzione che fornisce supporto a un linguaggio di programmazione, come Java, e di cui parleremo nel Paragrafo 15.1.2.

Per una migliore efficienza, come sarà chiaro più avanti, vogliamo che uno *heap* T abbia la minore altezza possibile e garantiamo questo requisito pretendendo che uno *heap* T soddisfi un’ulteriore proprietà strutturale: deve essere *completo*.

Proprietà di completezza di un albero binario: Uno *heap* T di altezza h è un albero binario *completo* se i livelli $0, 1, 2, \dots, h - 1$ di T hanno il massimo numero possibile di nodi (cioè il livello i ha 2^i nodi, per $0 \leq i \leq h - 1$) e i nodi del livello h si trovano nelle posizioni più a sinistra di quel livello.

L’albero della Figura 9.1 è completo perché i livelli 0, 1 e 2 sono pieni e i sei nodi del livello 3 sono nelle sei posizioni più a sinistra di quel livello. Per formalizzare ciò che intendiamo con “posizioni più a sinistra”, facciamo riferimento alla discussione sulla *numerazione per livelli* vista nel Paragrafo 8.3.2, nel contesto della rappresentazione di un albero binario mediante un array (e, in effetti, nel Paragrafo 9.3.2 parleremo dell’uso di un array per rappresentare uno *heap*). Un albero binario completo avente n elementi è un albero le cui posizioni sono numerate per livelli con i numeri che vanno da 0 a $n - 1$. Ad esempio, in

una rappresentazione mediante array dell'albero binario della Figura 9.1, le sue 13 entità sarebbero memorizzate in celle consecutive, da $A[0]$ a $A[12]$.

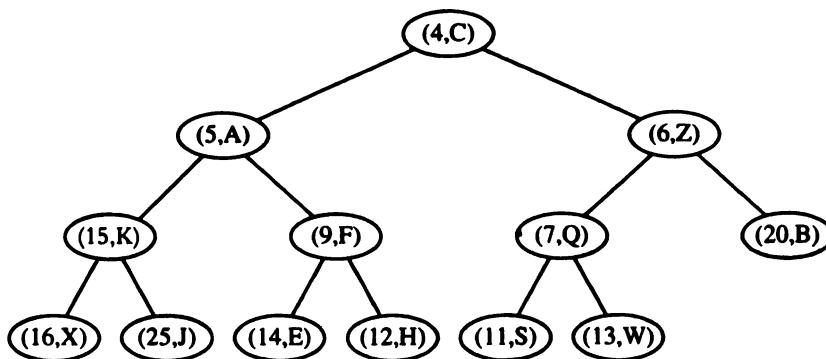


Figura 9.1: Esempio di heap che memorizza 13 entità con numeri interi come chiave. L'ultima posizione è quella che memorizza l'entità (13, W).

L'altezza di uno heap

Indichiamo con h l'altezza di T . Il fatto che T sia completo ha un'importante conseguenza, come si legge nella Proposizione 9.2.

Proposizione 9.2: *Uno heap T contenente n entità ha altezza $h = \lfloor \log n \rfloor$.*

Dimostrazione: Poiché T è completo, sappiamo che il numero di nodi dei livelli di T che vanno da 0 a $h - 1$ è esattamente uguale a $1 + 2 + 4 + \dots + 2^{h-1} = 2^h - 1$; inoltre, sappiamo che il numero di nodi posti al livello h è almeno 1 e al massimo 2^h . Quindi:

$$n \geq 2^h - 1 + 1 = 2^h \quad \text{e} \quad n \leq 2^h - 1 + 2^h = 2^{h+1} - 1$$

Prendendo il logaritmo di entrambi i membri della diseguaglianza $n \geq 2^h$, otteniamo $h \leq \log n$. Prendendo il logaritmo dell'altra diseguaglianza, $n \leq 2^{h+1} - 1$, e sistemandolo opportunamente gli addendi, otteniamo $h \geq \log(n + 1) - 1$. Dato che h è un numero intero, queste due diseguaglianze implicano che $h = \lfloor \log n \rfloor$. ■

9.3.2 Implementare una coda prioritaria con uno heap

La Proposizione 9.2 ha una conseguenza molto importante: implica che, se eseguiamo operazioni di aggiornamento di uno heap in un tempo proporzionale alla sua altezza, allora tali operazioni richiederanno un tempo logaritmico in funzione del numero di posizioni dell'albero. Occupiamoci, quindi, di come eseguire in modo efficiente i diversi metodi della coda prioritaria usando uno heap.

Per memorizzare coppie chiave-valore come voci di uno heap useremo lo schema di progettazione mediante composizione, già visto nel Paragrafo 9.2.1. I metodi `size` e `isEmpty` si possono implementare esaminando semplicemente l'albero, e anche l'operazione `min` è

banale, perché la proprietà di ordinamento di uno heap garantisce che l'elemento che si trova nella radice dell'albero è uno di quelli che hanno la chiave minima. Gli algoritmi interessanti sono quelli che implementano i metodi `insert` e `removeMin`.

Aggiunta di una voce a uno heap

Vediamo come si può eseguire il metodo `insert(k, v)` su una coda prioritaria implementata mediante uno heap T . Memorizziamo la coppia (k, v) sotto forma di voce in un nuovo nodo dell'albero. Per preservare la *proprietà di completezza dell'albero binario*, tale nuovo nodo può essere collocato soltanto nella posizione p subito alla destra del nodo più a destra del livello più basso dell'albero, oppure nella posizione più a sinistra di un nuovo livello, nel caso in cui l'ultimo livello sia pieno (o lo heap sia vuoto).

Risalita lungo lo heap dopo un inserimento

Dopo questa collocazione del nuovo nodo, l'albero T è ancora completo, ma può violare la *proprietà di ordinamento di uno heap*, quindi, a meno che la posizione p non sia la radice di T (cioè che la coda prioritaria fosse vuota prima dell'inserimento), confrontiamo la chiave che si trova nella posizione p con quella che si trova nel genitore di p , che indichiamo con q . Se $k_p \geq k_q$, la proprietà di ordinamento dello heap è soddisfatta e l'algoritmo termina. Se, invece, $k_p < k_q$, dobbiamo ripristinare la proprietà di ordinamento, obiettivo che localmente si può ottenere scambiando tra loro le voci che si trovano nelle posizioni p e q (come si può vedere nella Figura 9.2c e d). Questo scambio fa salire di un livello la nuova voce e, di nuovo, la proprietà di ordinamento dello heap potrebbe essere violata, per cui ripetiamo la procedura, risalendo in T finché non c'è più alcuna violazione della proprietà di ordinamento (come si può vedere nella Figura 9.2h).

Lo spostamento verso l'alto della voce appena inserita, realizzata mediante scambi, viene solitamente chiamato *up-heap bubbling* (cioè “bolle che risalgono verso la parte alta dello heap”). Uno scambio può risolvere la violazione della proprietà di ordinamento dello heap oppure propagarla di un livello verso l'alto all'interno dello heap. Nel caso peggiore, questa procedura sposta ripetutamente la nuova voce, fino a quando giunge nella radice dello heap T . Di conseguenza, nel caso peggiore, il numero di scambi eseguiti all'interno del metodo `insert` è uguale all'altezza di T e, per la Proposizione 9.2, questo valore è limitato da $\lfloor \log n \rfloor$.

Eliminazione di una voce avente chiave minima

Veniamo ora al metodo `removeMin` dell'ADT coda prioritaria. Sappiamo che nella radice r di T è sempre presente una voce avente la chiave minima (anche se ci possono essere più voci aventi la chiave minima), ma, in generale, non possiamo cancellare semplicemente il nodo r , perché questo ci lascerebbe due sottoalberi disconnessi.

Per garantire che la forma dello heap, dopo l'operazione di rimozione, rispetti la *proprietà di completezza di un albero binario*, dobbiamo cancellare la foglia che si trova nell'ultima posizione p di T , definita come la posizione più a destra del livello più profondo dell'albero. Per fare in modo che la voce presente nell'ultima posizione p rimanga all'interno della coda prioritaria, la copiamo nella radice r (al posto della voce avente la chiave minima, che deve essere eliminata). La Figura 9.3a e b mostra un esempio di queste fasi, dove la voce con chiave minima, (4, C), viene eliminata dalla radice e sostituita dalla voce (13, W) che si trovava nell'ultima posizione. A questo punto, il nodo che si trova nell'ultima posizione può essere rimosso dall'albero.

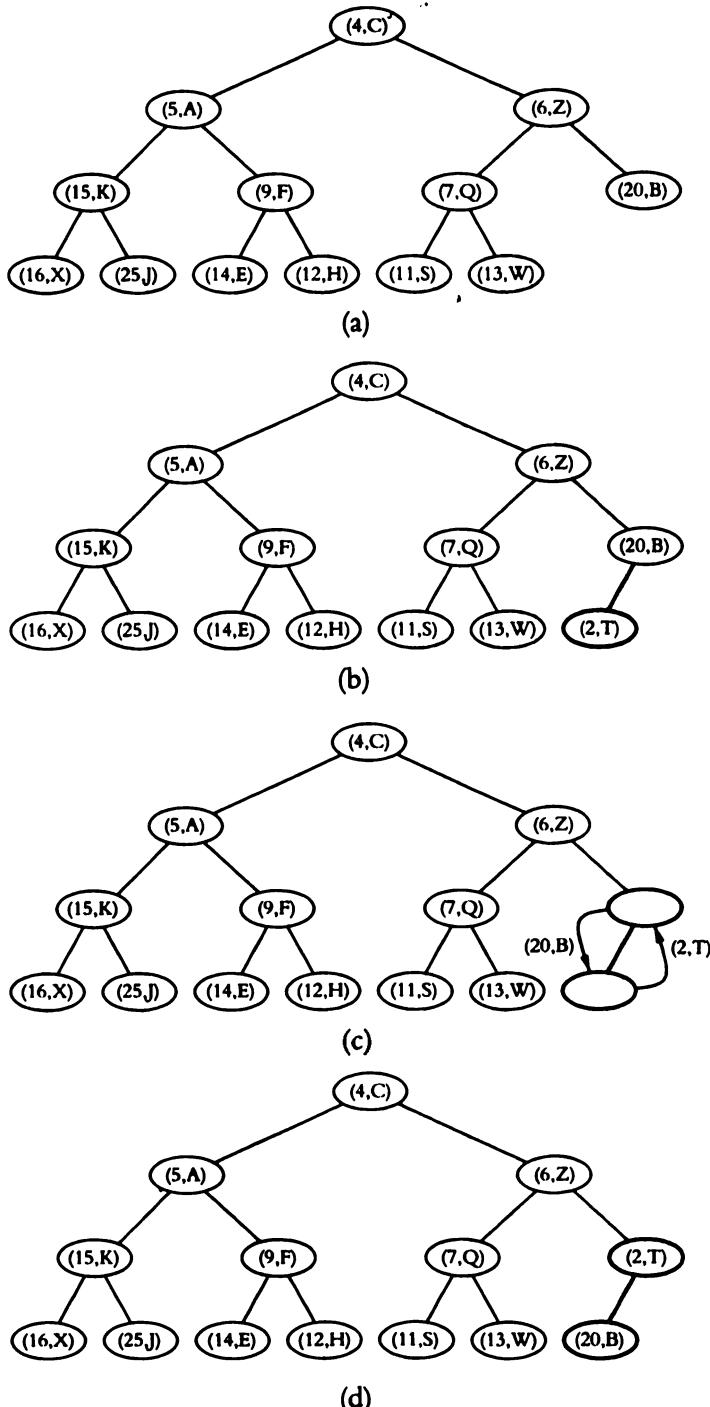
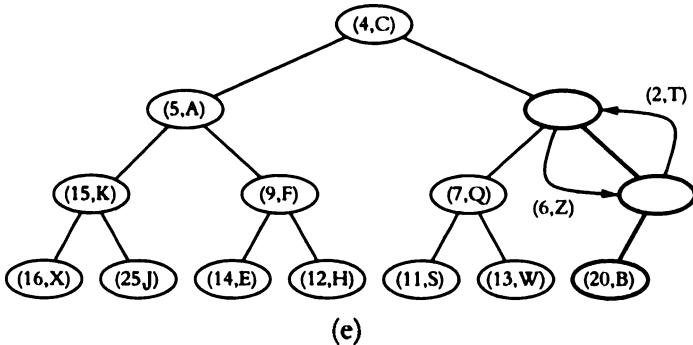
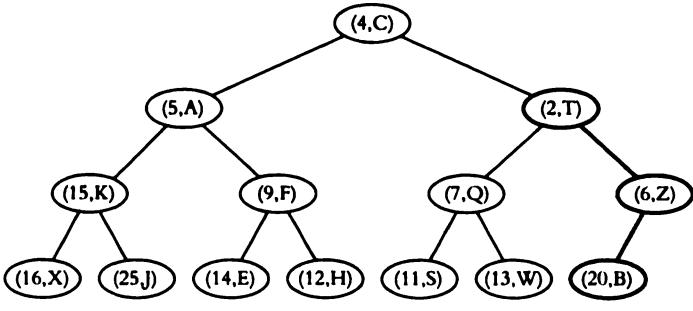


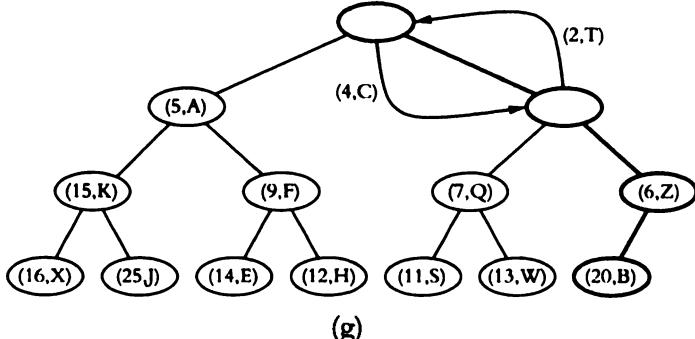
Figura 9.2 (prima parte): Inserimento di una nuova voce con chiave uguale a 2 nello heap della Figura 9.1: (a) heap iniziale; (b) dopo l'aggiunta di un nuovo nodo; (c, d) scambio per ripristinare localmente e in modo parziale la proprietà di ordinamento; (*continua*).



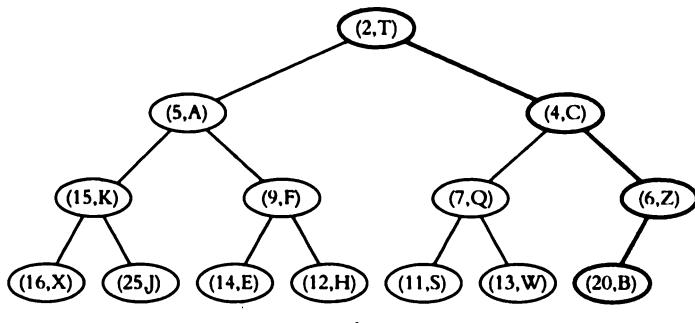
(e)



(f)



(g)



(h)

Figura 9.2 (seconda parte): (e, f) un altro scambio; (g, h) scambio conclusivo.

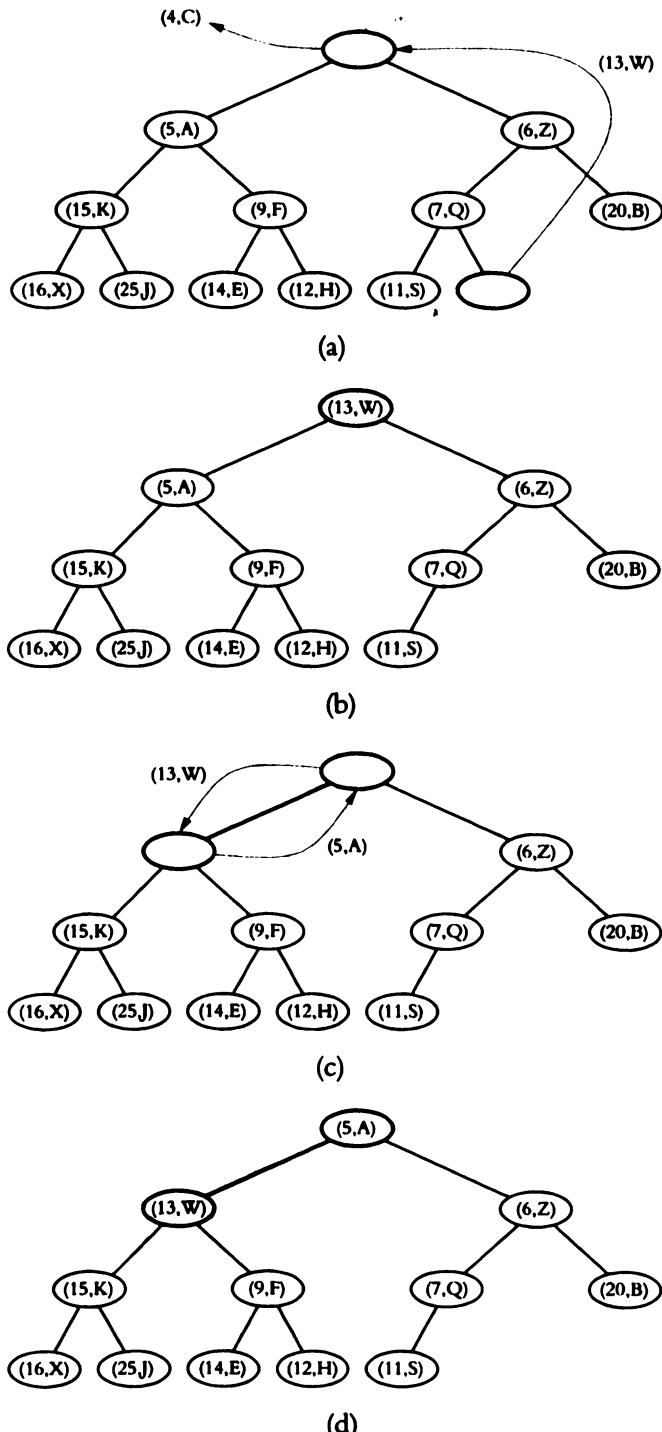
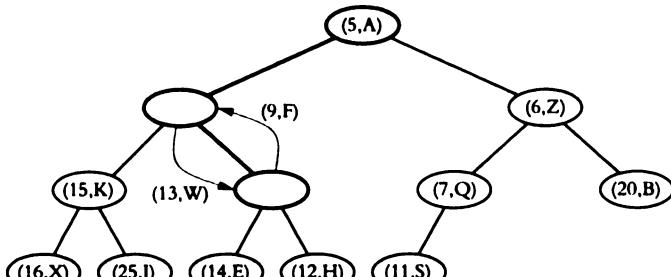
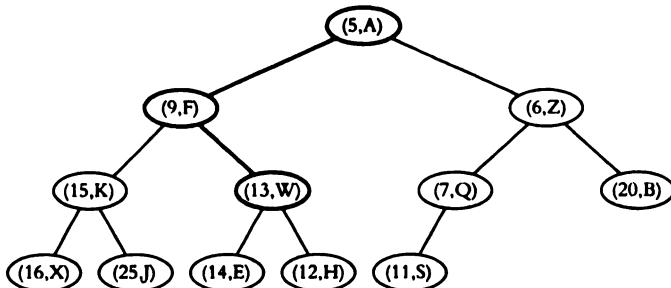


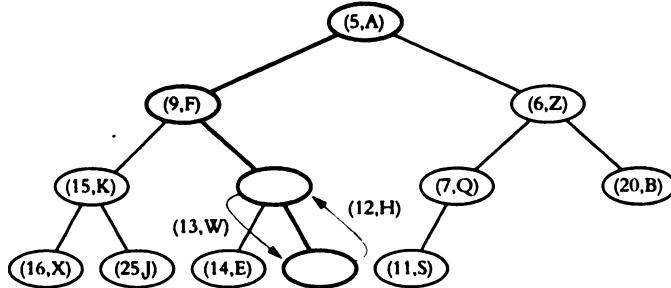
Figura 9.3 (prima parte): Rimozione da uno heap di una delle voci avente chiave minima:
(a, b) eliminazione dell'ultimo nodo, la cui voce viene memorizzata nella radice; **(c, d)** scambio per ripristinare localmente la proprietà di ordinamento; **(continua).**



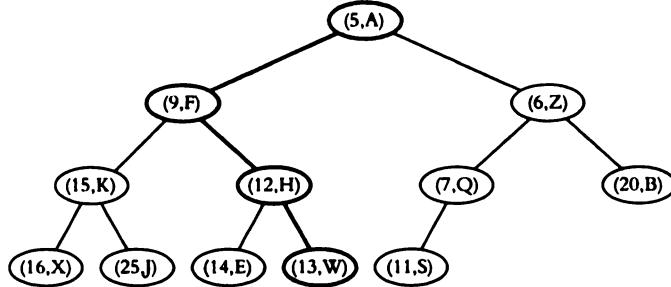
(e)



(f)



(g)



(h)

Figura 9.3 (seconda parte): (e, f) un altro scambio; (g, h) scambio conclusivo.

Discesa lungo lo heap dopo una rimozione

Tuttavia, non abbiamo ancora finito, perché, anche se ora T è completo, è probabile che violi la proprietà di ordinamento dello heap. Se T ha un solo nodo (la radice) allora la proprietà di ordinamento è banalmente rispettata e l'algoritmo termina. Altrimenti, distinguiamo due casi, partendo dalla posizione iniziale p che coincide con la radice di T .

- Se p non ha figlio destro, chiamiamo c il figlio sinistro di p .
- Altrimenti (p ha entrambi i figli), chiamiamo c il figlio di p avente chiave minore tra i due.

Se $k_p \leq k_c$, la proprietà di ordinamento dello heap è soddisfatta e l'algoritmo termina. Se, invece, $k_p > k_c$, dobbiamo ripristinare la proprietà di ordinamento dello heap, obiettivo che localmente si può raggiungere scambiando le voci memorizzate in p e in c (come si può vedere nella Figura 9.3c e d). È importante osservare che, quando p ha due figli, scegliamo appositamente il figlio avente chiave minore tra i due: in questo modo, non soltanto la chiave di c è minore della chiave di p , ma è anche non maggiore della chiave contenuta nel fratello di c . Questo garantisce che la proprietà di ordinamento dello heap viene localmente ripristinata quando la chiave di c viene portata in alto, perché è minore di quella che era in p (e che ora scende) ed è non maggiore di quella che si trova nella posizione del fratello di c prima della risalita.

Dopo aver ripristinato la proprietà di ordinamento del nodo p relativamente ai suoi figli, può essere ancora presente una violazione di tale proprietà in c , quindi può darsi che si debba continuare con gli scambi scendendo lungo l'albero, finché non si arriva al punto in cui non ci sono più violazioni della proprietà di ordinamento (si veda la seconda parte della Figura 9.3). Questo processo di scambio verso il basso è detto *down-heap bubbling* (cioè “bolle che scendono verso la parte bassa dello heap”). Ciascuno scambio risolve la violazione della proprietà di ordinamento oppure la propaga di un livello verso il basso all'interno dello heap. Nel caso peggiore, un'entità si sposta verso il basso fin quando è possibile (come si può vedere nella Figura 9.3), quindi il numero di scambi eseguiti all'interno del metodo `removeMin`, nel caso peggiore, è uguale all'altezza dello heap T , cioè, in base alla Proposizione 9.2, è $\lfloor \log n \rfloor$.

Rappresentazione con array di un albero binario completo

La rappresentazione di albero binario basata su array che abbiamo visto nel Paragrafo 8.3.2 è particolarmente adatta a un albero binario completo. Ricordiamo che in questa implementazione gli elementi dell'albero sono memorizzati in una lista A basata su array, in modo tale che l'elemento in posizione p venga memorizzato nella cella di A avente indice uguale al numero fornito dalla funzione di numerazione per livelli associato a p , cioè $f(p)$, definito in questo modo:

- Se p è la radice, allora $f(p) = 0$.
- Se p è il figlio sinistro della posizione q , allora $f(p) = 2f(q) + 1$.
- Se p è il figlio destro della posizione q , allora $f(p) = 2f(q) + 2$.

Per un albero completo di dimensione n , gli elementi hanno indici consecutivi appartenenti all'intervallo $[0, n - 1]$ e l'ultima posizione dell'albero ha sempre indice $n - 1$, come si può vedere nella Figura 9.4.

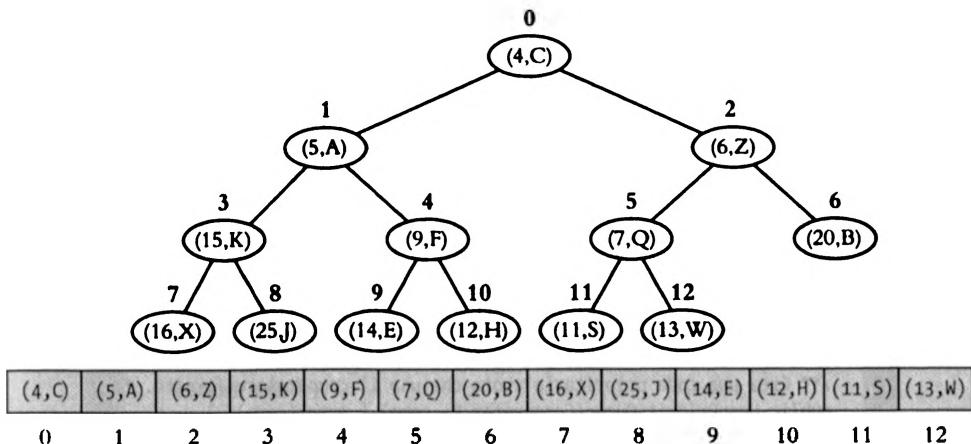


Figura 9.4: Rappresentazione di uno heap mediante un array.

Con la rappresentazione di heap basata su array si evitano alcune delle complicazioni derivanti dall'albero con struttura concatenata. In particolare, i metodi `insert` e `removeMin` devono individuare l'ultima posizione nello heap: con la rappresentazione basata su array, l'ultima posizione di uno heap avente dimensione n è semplicemente la cella di indice $n - 1$. Come si vedrà nell'Esercizio C-9.33, l'individuazione dell'ultima posizione in uno heap implementato mediante un albero a struttura concatenata richiede uno sforzo maggiore.

Se la dimensione che verrà assunta da una coda prioritaria non è nota a priori, l'uso di una rappresentazione basata su array rende necessario, di quando in quando, il ridimensionamento dinamico dell'array, come abbiamo visto per la classe `ArrayList` della libreria Java. Lo spazio utilizzato in memoria dalla rappresentazione basata su array di un albero binario completo avente n nodi è $O(n)$ e le prestazioni temporali che abbiamo visto per i metodi rimangono tali, anche se diventano *ammortizzate* (ricordando il Paragrafo 7.2.2).

Implementazione di uno heap in Java

Nel Codice 9.8 e 9.9 presentiamo un'implementazione in Java di una coda prioritaria basata su heap. Anche se immaginiamo che il nostro heap sia un albero binario, non usiamo l'ADT albero binario, perché preferiamo utilizzare la rappresentazione di albero basata su array, che è più efficiente, gestendo un esemplare di `ArrayList` (dalla libreria standard di Java) contenente le voci come oggetti composti. Per poter descrivere i nostri algoritmi usando la terminologia degli alberi, come *genitore*, *sinistro* e *destro*, la classe contiene metodi ausiliari `protected` che calcolano la numerazione per livelli del genitore o di un figlio di una determinata posizione (nelle righe del Codice 9.8 che vanno da 10 a 14). Tuttavia, in questa rappresentazione le "posizioni" sono semplicemente numeri interi usati come indici nella lista.

La nostra classe ha anche altri metodi ausiliari `protected` per effettuare gli spostamenti di voci all'interno della lista: `swap`, `upheap` e `downheap`. L'aggiunta di una nuova voce prevede il suo posizionamento alla fine della lista e il suo successivo eventuale spostamento mediante

upheap. Per eliminare la voce avente chiave minima che è memorizzata nella cella di indice 0, spostiamo in tale cella l'ultima voce della lista, che si trova in corrispondenza dell'indice $n - 1$, poi invochiamo downheap per riposizionarla correttamente.

Codice 9.8: Coda prioritaria che usa uno heap basato su array, estendendo la classe AbstractPriorityQueue descritta nel Codice 9.5 (prosegue nel Codice 9.9).

```

1  /** Un'implementazione di coda prioritaria mediante uno heap basato su array. */
2  public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3      /** Contenitore principale delle voci della coda prioritaria */
4      private ArrayList<Entry<K,V>> heap = new ArrayList<>();
5      /** Crea una coda prioritaria vuota che usa l'ordine naturale tra le chiavi. */
6      public HeapPriorityQueue() { super(); }
7      /** Crea una coda prioritaria vuota che usa il comparatore fornito. */
8      public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
9      // metodi ausiliari protected
10     protected int parent(int j) { return (j-1)/2; } // divisione con troncamento
11     protected int left(int j) { return 2*j + 1; }
12     protected int right(int j) { return 2*j + 2; }
13     protected boolean hasLeft(int j) { return left(j) < heap.size(); }
14     protected boolean hasRight(int j) { return right(j) < heap.size(); }
15     /** Scambia tra loro nella lista le voci di indice i e j. */
16     protected void swap(int i, int j) {
17         Entry<K,V> temp = head.get(i);
18         heap.set(i, heap.get(j));
19         heap.set(j, temp);
20     }
21     /** Sposta in alto la voce di indice j, se necessario per l'ordinamento. */
22     protected void upheap(int j) {
23         while (j > 0) { // prosegue fino alla radice (o a un break)
24             int p = parent(j);
25             if (compare(heap.get(j), heap.get(p)) >= 0) break; // ordinamento corretto
26             swap(j, p);
27             j = p;          // prosegue dalla posizione del genitore
28         }
29     }

```

Codice 9.9: Coda prioritaria che usa uno heap basato su array (continua dal Codice 9.8).

```

30    /** Sposta in basso la voce di indice j, se necessario per l'ordinamento. */
31    protected void downheap(int j) {
32        while (hasLeft(j)) { // prosegue verso il basso (o fino a un break)
33            int leftIndex = left(j);
34            int smallChildIndex = leftIndex; // il destro può ancora essere minore
35            if (hasRight(j)) {
36                int rightIndex = right(j);
37                if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
38                    smallChildIndex = rightIndex; // il figlio destro è minore
39            }
40            if (compare(heap.get(smallChildIndex), heap.get(i)) >= 0)
41                break;          // ordinamento corretto
42            swap(j, smallChildIndex);
43            j = smallChildIndex; // prosegue dalla posizione del figlio prescelto
44        }
45    }

```

```

47 // metodi pubblici
48 /** Restituisce il numero di voci presenti nella coda prioritaria. */
49 public int size() { return heap.size(); }
50 /** Restituisce una delle voci aventi chiave minima (senza rimuoverla). */
51 public Entry<K,V> min() {
52     if (heap.isEmpty()) return null;
53     return heap.get(0);
54 }
55 /** Inserisce una coppia chiave-valore e restituisce la voce creata. */
56 public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
57     checkKey(key); // metodo ausiliario di verifica (può lanciare eccezione)
58     Entry<K,V> newest = new PQEntry<>(key, value);
59     heap.add(newest); // aggiunge alla fine della lista
60     upheap(heap.size() - 1); // esegue up-heap per la voce appena aggiunta
61     return newest;
62 }
63 /** Elimina e restituisce una delle voci aventi chiave minima. */
64 public Entry<K,V> removeMin() {
65     if (heap.isEmpty()) return null;
66     Entry<K,V> answer = heap.get(0);
67     swap(0, heap.size() - 1); // sposta l'elemento minimo alla fine
68     heap.remove(heap.size() - 1); // e lo rimuove dalla lista
69     downheap(0); // quindi sistema la radice con down-heap
70     return answer;
71 }
72 }
```

9.3.3 Analisi di una coda prioritaria realizzata con uno heap

La Tabella 9.3 mostra i tempi d'esecuzione dei metodi dell'ADT coda prioritaria nel caso di implementazione mediante heap, nell'ipotesi che due chiavi possano essere confrontate in un tempo $O(1)$ e che lo heap T sia, a sua volta, implementato con un albero rappresentato mediante array o struttura concatenata.

In breve, tutti i metodi dell'ADT coda prioritaria vengono eseguiti in un tempo $O(1)$ o $O(\log n)$, dove n è il numero di voci presenti nella coda nel momento in cui il metodo viene eseguito. L'analisi del tempo d'esecuzione dei metodi si basa sulle seguenti osservazioni:

- Lo heap T ha n nodi, ciascuno dei quali memorizza un riferimento a una voce di tipo chiave-valore.
- L'altezza dello heap T è $O(\log n)$, perché T è completo (Proposizione 9.2).
- L'operazione `min` viene eseguita in un tempo $O(1)$ perché la radice dell'albero contiene un elemento adatto ad essere restituito.
- L'individuazione dell'ultima posizione di uno heap, operazione necessaria per i metodi `insert` e `removeMin`, viene eseguita in un tempo $O(1)$ nella rappresentazione basata su array e in un tempo $O(\log n)$ nella rappresentazione basata su albero concatenato (si veda, a questo proposito, l'Esercizio C-9.33).
- Nel caso peggiore, le procedure di up-heap e down-heap bubbling eseguono un numero di scambi uguale all'altezza dell'albero T .

Concludiamo dicendo che la struttura dati heap è una realizzazione molto efficiente del tipo di dato astratto "coda prioritaria", indipendentemente dal fatto che tale heap venga realizzato con una struttura concatenata o con un array. Diversamente dalle implementazioni

basate su una lista ordinata o non ordinata, viste in precedenza, l'implementazione di coda prioritaria basata su heap raggiunge tempi d'esecuzione rapidi tanto per gli inserimenti quanto per le rimozioni.

Tabella 9.3: Prestazioni di una coda prioritaria realizzata mediante uno heap. Indichiamo con n il numero di voci presenti nella coda prioritaria nel momento in cui viene eseguita l'operazione. Lo spazio utilizzato è $O(n)$. Nel caso di una rappresentazione basata su array, i tempi d'esecuzione delle operazioni `min` e `removeMin` sono ammortizzati, per effetto dei ridimensionamenti dell'array dinamico; gli stessi limiti costituiscono il caso peggiore nella realizzazione mediante albero concatenato.

Metodo	Tempo d'esecuzione
<code>size</code> , <code>isEmpty</code>	$O(1)$
<code>min</code>	$O(1)$
<code>insert</code>	$O(\log n)^*$
<code>removeMin</code>	$O(\log n)^*$

*ammortizzato, usando un array dinamico

9.3.4 Costruzione di uno heap dal basso verso l'alto (*bottom-up*)*

Partendo da uno heap inizialmente vuoto, n invocazioni consecutive dell'operazione `insert` verranno eseguite in un tempo complessivo $O(n \log n)$, nel caso peggiore. Tuttavia, se tutte le n coppie chiave-valore da inserire nello heap sono note fin dall'inizio, come avviene durante la prima fase dell'algoritmo *heap-sort* (ordinamento mediante heap, che presenteremo nel Paragrafo 9.4.2), esiste un metodo di costruzione alternativo, che procede dal basso verso l'alto (*bottom-up heap construction*) e viene eseguito in un tempo complessivo $O(n)$.

In questo paragrafo descriveremo la costruzione di uno heap procedendo dal basso verso l'alto e implementeremo tale procedura, che può essere usata come costruttore di una coda prioritaria basata su heap.

Per semplicità di esposizione, descriviamo la procedura ipotizzando che il numero di chiavi, n , sia un numero intero tale che $n = 2^{h+1} - 1$, cioè assumiamo che lo heap sia un albero binario completo con tutti i livelli pieni, in modo che la sua altezza sia $h = \log(n + 1) - 1$. Immaginandola in modo non ricorsivo, la costruzione di uno heap in modalità *bottom-up* consiste delle seguenti $h + 1 = \log(n + 1)$ fasi.

- Nella prima fase (Figura 9.5b) costruiamo $(n + 1)/2$ heap elementari, ciascuno dei quali memorizza una sola voce.
- Nella seconda fase (Figura 9.5c e d), componiamo $(n + 1)/4$ heap, ciascuno dei quali memorizza tre voci, unendo coppie di heap elementari e aggiungendo a ciascuna coppia una nuova voce, che viene posta inizialmente nella radice e può essere, poi, scambiata con la voce memorizzata in uno dei suoi figli, per ripristinare la proprietà di ordinamento dello heap.
- Nella terza fase (Figura 9.5e e f), componiamo $(n + 1)/8$ heap, ciascuno dei quali memorizza 7 voci, unendo coppie di heap aventi 3 voci ciascuno (costruiti nella fase precedente) e aggiungendo a ciascuna coppia una nuova voce, che viene posta

* Usiamo un asterisco per segnalare paragrafi del libro che contengono materiale più avanzato di quello trattato nella restante parte del capitolo; si tratta di materiale che, in una prima lettura, può essere considerato facoltativo.

inizialmente nella radice, ma può darsi che debba scendere verso il basso per ripristinare la proprietà di ordinamento, come determinato dalla procedura di down-heap bubbling.

- i. Nella generica, i -esima fase, con $2 \leq i \leq h$, componiamo $(n + 1)/2^i$ heap, ciascuno dei quali memorizza $2^i - 1$ voci, unendo coppie di heap aventi $(2^{i-1} - 1)$ voci ciascuno (costruiti nella fase precedente) e aggiungendo a ciascuna coppia una nuova voce che viene posta inizialmente nella radice, ma può darsi che debba scendere verso il basso per ripristinare la proprietà di ordinamento, come determinato dalla procedura di down-heap bubbling.

- $h + 1$. Nell'ultima fase (Figura 9.5g e h), componiamo lo heap conclusivo, che memorizza le n voci, unendo due heap aventi $(n - 1)/2$ voci ciascuno (costruiti nella fase precedente) e aggiungendo una nuova voce, che viene posta inizialmente nella radice, ma può darsi che debba scendere verso il basso per ripristinare la proprietà di ordinamento, come determinato dalla procedura di down-heap bubbling.

La Figura 9.5 mostra la costruzione *bottom-up* di uno heap con $h = 3$.

Implementazione in Java della costruzione *bottom-up* di uno heap

L'implementazione della costruzione *bottom-up* di uno heap è abbastanza semplice, vista l'esistenza del metodo ausiliario che esegue la procedura di down-heap bubbling. La "fusione" (*merge*) di due heap aventi la stessa dimensione che siano sottoalberi di una medesima posizione p , come descritto all'inizio di questo paragrafo, può essere effettuata semplicemente applicando la procedura *down-heap* all'entità presente in p . Questo è proprio ciò che accade, ad esempio, alla chiave 14 passando dalla Figura 9.5f alla Figura 9.5g.

Usando la nostra rappresentazione di heap basata su array, se iniziamo memorizzando tutte le n entità nell'array in ordine arbitrario, possiamo implementare la procedura di costruzione bottom-up dello heap con un unico ciclo che invoca *downheap* per ciascuna posizione dell'albero, a patto che tali invocazioni avvengano seguendo un ordine che parte dal livello più profondo e termina nella radice dell'albero. In effetti, il ciclo può iniziare dalla posizione interna più profonda, dal momento che invocare la procedura di down-heap bubbling su una foglia non ha alcun effetto.

Nel Codice 9.10 aggiungiamo alla classe `HeapPriorityQueue` già definita nel Paragrafo 9.3.2 il supporto per la costruzione bottom-up a partire da un contenitore fornito come parametro. Aggiungiamo anche un metodo ausiliario non pubblico, `heapify` (cioè "trasforma in uno heap"), che invoca `downheap` per ogni posizione che non sia una foglia, a partire da quelle più profonde e terminando con la radice.

Aggiungiamo, quindi, un costruttore a quelli già presenti nella classe, che accetti una sequenza iniziale di chiavi e valori, sotto forma di due array che si suppone abbiano la stessa lunghezza. Creiamo le voci accoppiando la prima chiave con il primo valore, la seconda chiave con il secondo valore, e così via. Poi invochiamo il metodo ausiliario `heapify` perché sia garantita la proprietà di ordinamento dello heap. Per brevità abbiamo omesso di inserire un analogo costruttore che accetti un comparatore da usare nella coda prioritaria invece di considerare l'ordinamento naturale tra le chiavi.

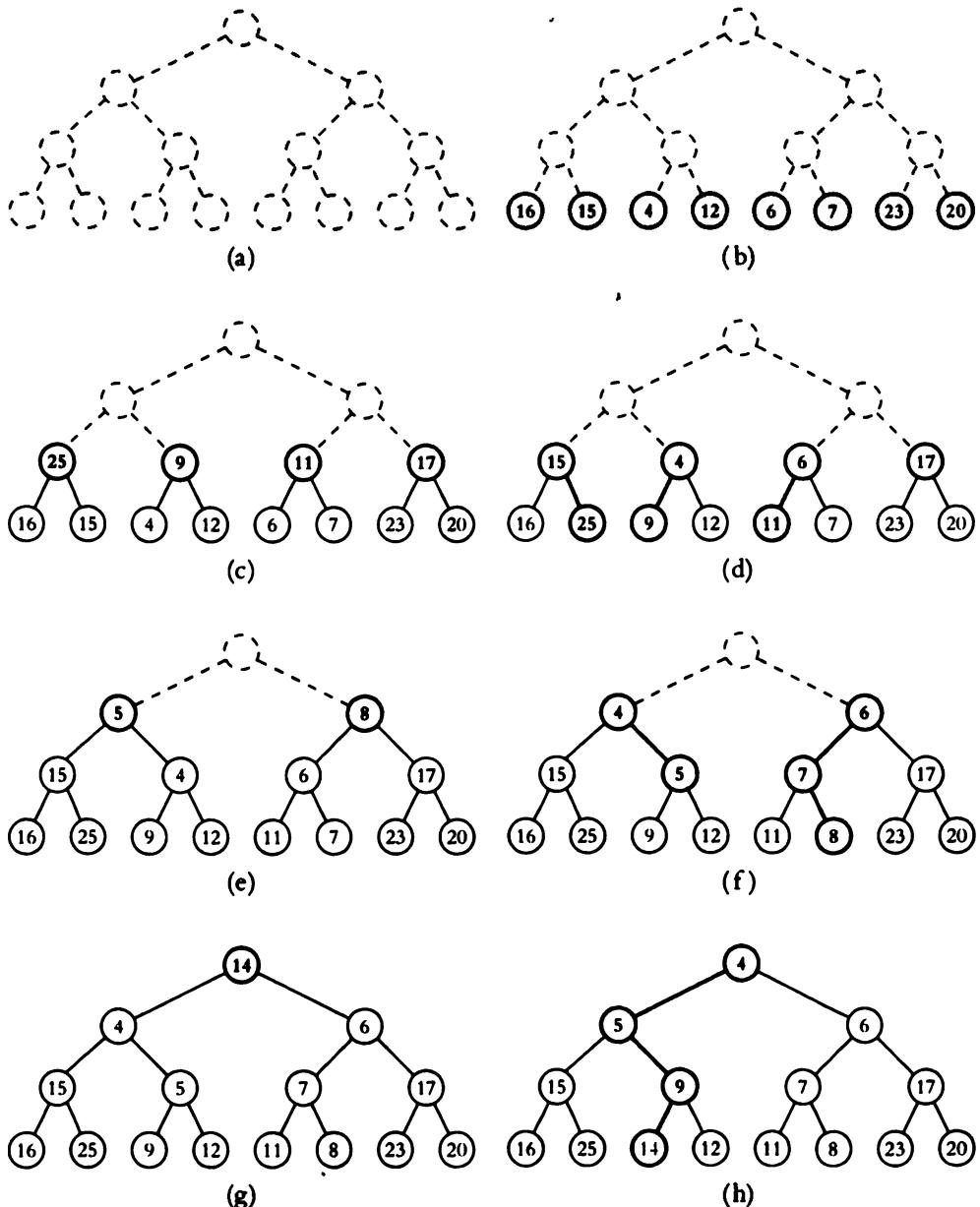


Figura 9.5: Costruzione *bottom-up* di uno heap con 15 voci: (a, b) iniziamo con la costruzione di heap aventi una sola voce, al livello più basso; (c, d) combiniamo questi heap per formare heap aventi 3 voci; (e, f) costruiamo heap con 7 voci; (g, h) creiamo lo heap conclusivo. I percorsi seguiti dalle procedure di down-heap bubbling sono evidenziati nelle figure d, e f e h. Per semplicità, all'interno di ogni nodo abbiamo riportato soltanto le chiavi, invece delle voci complete.

Codice 9.10: Codice da aggiungere alla classe HeapPriorityQueue (definita nel Codice 9.8 e 9.9) per fornire supporto alla costruzione di uno heap in un tempo lineare quando sia dato un insieme iniziale di coppie chiave-valore.

```

1  /** Crea una coda prioritaria contenente le coppie chiave-valore fornite. */
2  public HeapPriorityQueue(K[] keys, V[] values) {
3      super();
4      for (int j=0; j < Math.min(keys.length, values.length); j++)
5          heap.add(new PQEntry<>(keys[j], values[j]));
6      heapify();
7  }
8  /** Effettua la costruzione bottom-up dello heap in un tempo lineare. */
9  protected void heapify() {
10     int startIndex = parent(size()-1); // inizia dal genitore dell'ultima entità
11     for (int j=startIndex; j >= 0; j--) // continua fino alla radice
12         downheap(j);
13 }
```

Analisi asintotica della costruzione bottom-up di uno heap

La costruzione bottom-up di uno heap è asintoticamente più veloce dell'inserimento progressivo di n voci, una dopo l'altra, in uno heap inizialmente vuoto. Intuitivamente, viene eseguita un'unica operazione di down-heap bubbling per ogni posizione dell'albero, invece di un'unica operazione di up-heap, sempre per ogni posizione: dato che il numero di nodi vicini alla parte bassa dell'albero è decisamente maggiore del numero di nodi che si trovano nella parte alta, la somma delle lunghezze dei percorsi verso il basso è lineare, come enunciato dalla seguente proprietà.

Proposizione 9.3: La costruzione bottom-up di uno heap avente n entità richiede un tempo $O(n)$, nell'ipotesi che due chiavi possano essere confrontate tra loro in un tempo $O(1)$.

Dimostrazione: Il costo principale della costruzione è dovuto alle fasi di down-heap bubbling eseguite a partire da ciascuna posizione che non sia una foglia. Indichiamo con π_v il percorso di T che va dal nodo interno v alla foglia che lo segue nell'attraversamento in ordine simmetrico: quindi, il percorso π_v parte da v , passa per il figlio destro di v e, poi, procede verso sinistra finché non raggiunge una foglia. Anche se π_v non è necessariamente il percorso seguito dalla fase di down-heap bubbling eseguita per v , il suo numero di rami, $\|\pi_v\|$, è proporzionale all'altezza del sottoalbero avente radice v e, quindi, rappresenta il valore limite superiore per la complessità dell'operazione di down-heap per v . Il tempo d'esecuzione totale dell'algoritmo di costruzione bottom-up di uno heap è, quindi, limitato superiormente dalla somma $\sum_v \|\pi_v\|$. Basandosi sull'intuizione, la Figura 9.6 illustra questa dimostrazione in modo "grafico", contrassegnando ciascun ramo dell'albero con un'etichetta che corrisponde al nodo interno v il cui percorso π_v contiene quel ramo.

Affermiamo che i percorsi π_v per tutti i nodi interni v sono distinti per quanto riguarda i rami che li compongono, quindi la somma delle lunghezze di tali percorsi è limitata superiormente dal numero totale di rami dell'albero, che è $O(n)$. Per dimostrare questa affermazione, definiamo i termini "ramo diretto verso destra" e "ramo diretto verso sinistra" per indicare un ramo che collega un genitore con il suo figlio destro o, rispettivamente, sinistro. Un ramo e diretto verso destra può appartenere solamente al percorso π_v , associato al nodo v che, nella relazione rappresentata da e , svolge il ruolo di genitore. I rami diretti

verso sinistra, invece, possono essere suddivisi sulla base della foglia in cui si giungerebbe proseguendo verso sinistra fino a raggiungere una foglia. Ogni nodo interno v usa, nel proprio percorso π_v , soltanto quei rami diretti verso sinistra che appartengono al gruppo di rami che portano alla foglia che segue immediatamente v nell'attraversamento in ordine simmetrico. Dato che ogni nodo interno deve necessariamente avere una diversa foglia come immediato successore nell'attraversamento in ordine simmetrico, non possono esistere due percorsi che contengono lo stesso ramo diretto verso sinistra. Possiamo, quindi, concludere che la costruzione bottom-up dello heap T richiede un tempo $O(n)$. ■

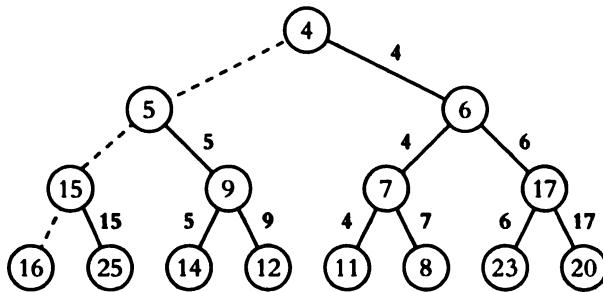


Figura 9.6: Dimostrazione grafica del tempo d'esecuzione lineare per la costruzione bottom-up di uno heap. Ogni ramo è stato etichettato con la chiave del nodo v (se esiste) tale che π_v contenga v .

9.3.5 Utilizzo della classe `java.util.PriorityQueue`

Nella libreria standard di Java non esiste un'interfaccia che definisca una coda prioritaria, ma c'è una classe, `java.util.PriorityQueue`, che implementa l'interfaccia `java.util.Queue` e che, invece di inserire e rimuovere elementi secondo la normale strategia FIFO usata dalla maggior parte delle code, elabora i propri elementi sulla base di priorità. L'*inizio (front)* della coda sarà sempre un elemento di priorità minima, con le priorità che si basano sull'ordinamento naturale degli elementi o su un oggetto comparatore fornito come parametro nel momento in cui si è costruita la coda prioritaria.

La differenza più significativa tra la classe `java.util.PriorityQueue` e il nostro ADT coda prioritaria è il modello con cui vengono gestite le chiavi e i valori. Mentre la nostra interfaccia pubblica distingue tra chiavi e valori, la classe `java.util.PriorityQueue` si basa su elementi di un solo tipo: l'elemento viene, a tutti gli effetti, trattato come se fosse la chiave.

Se un utilizzatore di `java.util.PriorityQueue` vuole inserire chiavi e valori separati, è suo compito definire e inserire nel contenitore appositi oggetti composti, garantendo che tali oggetti vengano confrontati sulla base delle loro chiavi. Il Java Collections Framework definisce una propria interfaccia per questo tipo di oggetti composti, `java.util.Map.Entry`, con un'implementazione concreta nella classe `java.util.AbstractMap.SimpleEntry` (il prossimo capitolo si occuperà della mappa come tipo di dato astratto).

La Tabella 9.4 mostra le corrispondenze tra i metodi del nostro ADT coda prioritaria e quelli della classe `java.util.PriorityQueue`. La classe `java.util.PriorityQueue` è implementata con uno heap, per cui garantisce prestazioni temporali $O(\log n)$ per i metodi `add` e `remove`, e prestazioni tempo-costanti per i metodi `d'accesso peek`, `size` e `isEmpty`. Mette inoltre a disposizione un metodo che riceve un parametro, `remove(e)`, il quale elimina dalla coda prioritaria

l'elemento specificato, e, anche se tale metodo viene eseguito in un tempo $O(n)$, eseguendo una ricerca sequenziale per individuare l'elemento all'interno dello heap (nel Paragrafo 9.5 estenderemo la nostra implementazione di coda prioritaria basata su heap in modo che consenta una più efficiente rimozione di una voce arbitraria, così come l'aggiornamento di una voce già presente nella coda prioritaria).

Tabella 9.4: Metodi del nostro ADT coda prioritaria e i corrispondenti metodi della classe `java.util.PriorityQueue`.

Il nostro ADT coda prioritaria	La classe <code>java.util.PriorityQueue</code>
<code>insert(k,v)</code>	<code>add(new SimpleEntry(k,v))</code>
<code>min()</code>	<code>peek()</code>
<code>removeMin()</code>	<code>remove()</code>
<code>size()</code>	<code>size()</code>
<code>isEmpty()</code>	<code>isEmpty()</code>

9.4 Ordinare con una coda prioritaria

L'ordinamento è una delle applicazioni delle code prioritarie: ci viene data una sequenza di elementi che possono essere confrontati tra loro secondo una relazione d'ordine totale e vogliamo sistemerla in modo che gli elementi vi figurino in ordine crescente (o almeno non decrescente, nel caso in cui ci siano elementi duplicati). L'algoritmo per ordinare una sequenza S usando una coda prioritaria P è abbastanza semplice ed è costituito dalle due fasi seguenti:

1. Nella prima fase inseriamo gli elementi di S come chiavi in una coda prioritaria P inizialmente vuota, eseguendo n operazioni `insert`, una per ogni elemento.
2. Nella seconda fase estraiamo gli elementi da P in ordine non decrescente, eseguendo n operazioni `removeMin`, per memorizzarli di nuovo in S nell'ordine in cui vengono estratti.

Il Codice 9.11 riporta un'implementazione di questo algoritmo in Java, ipotizzando che la sequenza sia memorizzata in una lista posizionale (il codice per un contenitore di tipo diverso, come un array o, in generale, una lista con indice, sarebbe del tutto analogo).

Codice 9.11: Implementazione del metodo `pqSort` che ordina gli elementi di una lista posizionale usando una coda prioritaria inizialmente vuota per produrre l'ordinamento richiesto.

```

1  /** Ordina la sequenza S, usando una coda prioritaria P inizialmente vuota. */
2  public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?> P) {
3      int n = S.size();
4      for (int j=0; j < n; j++) {
5          E element = S.remove(S.first());
6          P.insert(element, null);    // element è la chiave, null è il valore
7      }
8      for (int j=0; j < n; j++) {
9          E element = P.removeMin().getKey();
10         S.addLast(element);    // la chiave minima di P viene trasferita alla fine di S
11     }
12 }
```

L'algoritmo funziona correttamente con qualsiasi coda prioritaria P , indipendentemente da come sia implementata, ma il tempo d'esecuzione dell'algoritmo è determinato dai tempi d'esecuzione delle operazioni `insert` e `removeMin`, che dipendono, a loro volta, da come è stata implementata P . In realtà, il metodo `pqSort` dovrebbe essere considerato più uno "schema" di ordinamento che un "algoritmo", perché non specifica l'implementazione della coda prioritaria P . Lo schema `pqSort` è il paradigma seguito da diversi algoritmi di ordinamento ben noti, tra i quali l'ordinamento per selezione (*selection sort*), l'ordinamento per inserimento (*insertion sort*) e l'ordinamento mediante heap (*heap sort*), di cui parleremo nel prossimo paragrafo.

9.4.1 Ordinamento per selezione e ordinamento per inserimento

Per iniziare, vediamo come lo schema `pqSort` dia luogo a due classici algoritmi di ordinamento quando usi una coda prioritaria implementata mediante una lista non ordinata o una lista ordinata.

Ordinamento per selezione

Nella Fase 1 dello schema `pqSort` inseriamo tutti gli elementi in una coda prioritaria P ; nella Fase 2 estraiamo ripetutamente l'elemento minimo da P usando il metodo `removeMin`. Se implementiamo P con una lista non ordinata, allora la Fase 1 di `pqSort` richiede un tempo $O(n)$, perché possiamo inserire ciascun elemento in un tempo $O(1)$. Nella Fase 2, il tempo d'esecuzione di ciascuna operazione `removeMin` è proporzionale alla dimensione di P , quindi il collo di bottiglia computazionale è la "selezione" ripetuta dell'elemento minimo, che avviene nella Fase 2. Per questo motivo, questo algoritmo è meglio conosciuto con il nome di *ordinamento per selezione* (*selection sort*) e il suo funzionamento è illustrato dalla Figura 9.7.

		<i>Sequenza S</i>	<i>Coda prioritaria P</i>
Inizio		(7, 4, 8, 2, 5, 3, 9)	()
Fase 1	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(7, 4)
	:	:	:
	(g)	()	(7, 4, 8, 2, 5, 3, 9)
Fase 2	(a)	(2)	(7, 4, 8, 5, 3, 9)
	(b)	(2, 3)	(7, 4, 8, 5, 9)
	(c)	(2, 3, 4)	(7, 8, 5, 9)
	(d)	(2, 3, 4, 5)	(7, 8, 9)
	(e)	(2, 3, 4, 5, 7)	(8, 9)
	(f)	(2, 3, 4, 5, 7, 8)	(9)
	(g)	(2, 3, 4, 5, 7, 8, 9)	()

Figura 9.7: Esecuzione dell'ordinamento per selezione sulla sequenza $S = (7, 4, 8, 2, 5, 3, 9)$.

Come già osservato, il collo di bottiglia è nella Fase 2, dove eliminiamo ripetutamente dalla coda prioritaria P una delle voci aventi chiave minima. La dimensione di P parte dal valore iniziale n e diminuisce di un'unità dopo ogni operazione `removeMin`, fino a diventare

zero, quindi la prima operazione `removeMin` richiede un tempo $O(n)$, la seconda un tempo $O(n - 1)$ e così via, fino alla n -esima e ultima operazione, che viene eseguita in un tempo $O(1)$. Perciò il tempo totale richiesto per eseguire la Fase 2 è:

$$O(n + (n - 1) + \dots + 2 + 1) = O\left(\sum_{i=1}^n i\right)$$

Per la Proposizione 4.3, la sommatoria di destra vale $n(n+1)/2$, per cui la Fase 2 richiede un tempo d'esecuzione $O(n^2)$, tempo richiesto quindi anche dall'intero algoritmo di ordinamento per selezione.

Ordinamento per inserimento

Se implementiamo la coda prioritaria P usando una lista ordinata, il tempo d'esecuzione della Fase 2 migliora e diventa $O(n)$, perché ciascuna operazione `removeMin` su P richiede ora un tempo $O(1)$. Sfortunatamente, però, il collo di bottiglia del tempo d'esecuzione si sposta nella Fase 1, perché, nel caso peggiore, ogni operazione `insert` necessita di un tempo proporzionale alla dimensione di P . Questo algoritmo di ordinamento è noto con il nome di *ordinamento per inserimento* (*insertion sort*), e il suo funzionamento è illustrato dalla Figura 9.8, perché il suo punto critico riguarda il ripetuto "inserimento" di un nuovo elemento nella giusta posizione di una lista ordinata.

		<i>Sequenza S</i>	<i>Coda prioritaria P</i>
Inizio		(7, 4, 8, 2, 5, 3, 9)	()
Fase 1	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(4, 7)
	(c)	(2, 5, 3, 9)	(4, 7, 8)
	(d)	(5, 3, 9)	(2, 4, 7, 8)
	(e)	(3, 9)	(2, 4, 5, 7, 8)
	(f)	(9)	(2, 3, 4, 5, 7, 8)
	(g)	()	(2, 3, 4, 5, 7, 8, 9)
Fase 2	(a)	(2)	(3, 4, 5, 7, 8, 9)
	(b)	(2, 3)	(4, 5, 7, 8, 9)
	:	:	:
	(g)	(2, 3, 4, 5, 7, 8, 9)	()

Figura 9.8: Esecuzione dell'ordinamento per inserimento sulla sequenza $S = (7, 4, 8, 2, 5, 3, 9)$. Nella Fase 1 eliminiamo ripetutamente il primo elemento di S e lo inseriamo in P . Nella Fase 2 eseguiamo ripetutamente l'operazione `removeMin` su P e aggiungiamo alla fine di S l'elemento restituito.

Analizzando il tempo d'esecuzione della Fase 1 dell'ordinamento per inserimento, osserviamo che è:

$$O(n + (n - 1) + \dots + 2 + 1) = O\left(\sum_{i=1}^n i\right)$$

Di nuovo, ricordando la Proposizione 4.3, la Fase 1 viene eseguita in un tempo $O(n^2)$ e, quindi, lo stesso accade per l'intero algoritmo di ordinamento per inserimento.

In alternativa, potremmo cambiare la nostra definizione di ordinamento per inserimento in modo che, nella Fase 1, gli elementi vengano inseriti nella lista che implementa la coda prioritaria a partire dalla fine, nel qual caso l'esecuzione dell'ordinamento per inserimento su una sequenza che sia già ordinata richiederebbe un tempo $O(n)$. Nel caso più generale, il tempo d'esecuzione dell'ordinamento per inserimento diventerebbe $O(n + I)$, dove I è il numero di *inversioni* presenti nella sequenza iniziale, cioè è il numero di coppie di elementi che, nella sequenza iniziale, non si trovano nell'ordine corretto tra loro.

9.4.2 Heap Sort

Come già detto, realizzando una coda prioritaria con uno heap si ha un vantaggio: tutti i metodi dell'ADT coda prioritaria vengono eseguiti in un tempo logaritmico o migliore. Questa realizzazione, quindi, è adatta a quelle applicazioni che cercano tempi d'esecuzione rapidi per tutti i metodi della coda prioritaria. Torniamo nuovamente allo schema `pqSort`, usando questa volta una coda prioritaria implementata mediante un array.

Durante la Fase 1, la i -esima operazione `insert` richiede un tempo $O(\log i)$, perché dopo l'esecuzione dell'operazione lo heap contiene i entità. Questa fase richiede, quindi, un tempo complessivo $O(n \log n)$, che può essere migliorato e diventa $O(n)$ se si usa la costruzione bottom-up descritta nel Paragrafo 9.3.4.

Durante la seconda fase del metodo `pqSort`, la j -esima operazione `removeMin` viene eseguita in un tempo $O(\log(n - j + 1))$, perché nel momento in cui inizia l'operazione lo heap contiene $n - j + 1$ entità. Sommando per tutti i valori di j , questa fase richiede nuovamente un tempo $O(n \log n)$, per cui l'intero algoritmo di ordinamento che usa una coda prioritaria realizzata mediante heap viene eseguito in un tempo $O(n \log n)$. Questo algoritmo di ordinamento è noto con il nome di *ordinamento a heap (heap sort)* e le sue prestazioni sono riassunte dalla proposizione seguente.

Proposizione 9.4: *L'algoritmo heap sort ordina una sequenza S di n elementi in un tempo $O(n \log n)$, nell'ipotesi che due elementi di S possano essere confrontati in un tempo $O(1)$.*

Vale forse la pena di osservare che il tempo d'esecuzione $O(n \log n)$ di heap sort è significativamente migliore del tempo d'esecuzione $O(n^2)$ degli algoritmi di ordinamento per selezione e per inserimento.

Implementare heap sort sul posto

Se la sequenza S da ordinare è implementata mediante una lista basata su array, come un esemplare di `ArrayList` in Java, possiamo velocizzare heap sort e ridurre i suoi requisiti di spazio in memoria di un fattore costante usando una porzione dell'array stesso per memorizzare lo heap, evitando così l'utilizzo di una struttura dati ausiliaria per lo heap. Occorre però modificare l'algoritmo in questo modo:

1. Ridefiniamo le operazioni dello heap in modo che sia *orientato verso il valore massimo*, con ogni chiave che è non maggiore dei suoi figli. Questo obiettivo può essere raggiunto modificando il codice degli algoritmi, oppure fornendo un comparatore che inverta

il risultato di ogni confronto. In ogni momento, durante l'esecuzione dell'algoritmo, usiamo la parte sinistra di S , fino a un certo indice $i - 1$, per memorizzare le voci dello heap, mentre la parte destra, dall'indice i all'indice $n - 1$, memorizza gli elementi della sequenza. Di conseguenza, le prime i celle di S (aventi indici $0, \dots, i - 1$) costituiscono la lista con indice che rappresenta lo heap.

2. Nella prima fase dell'algoritmo, partiamo da uno heap vuoto e spostiamo il confine tra lo heap e la sequenza procedendo da sinistra verso destra, un passo alla volta. Al passo i (con $i = 1, \dots, n$) espandiamo lo heap aggiungendovi l'elemento memorizzato nella cella di indice $i - 1$.
3. Nella seconda fase dell'algoritmo, partiamo da una sequenza vuota e spostiamo il confine tra lo heap e la sequenza procedendo da destra verso sinistra, un passo alla volta. Al passo i (con $i = 1, \dots, n$) eliminiamo l'elemento massimo dallo heap e lo memorizziamo nella cella di indice $n - i$.

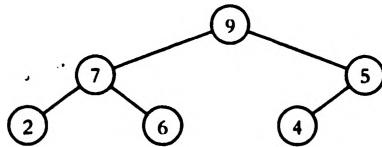
In generale, diciamo che un algoritmo di ordinamento opera *sul posto* (*in-place*) se usa soltanto una piccola quantità di memoria oltre a quella dedicata alla sequenza che memorizza gli oggetti da ordinare. La variante di heap sort appena descritta opera sul posto: invece di trasferire gli elementi al di fuori della sequenza, per poi riportarveli, semplicemente li spostiamo al suo interno. La Figura 9.9 mostra la seconda fase dell'algoritmo heap sort, nella variante che opera sul posto.

9.5 Code prioritarie flessibili

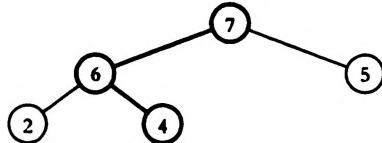
I metodi dell'ADT coda prioritaria che abbiamo elencato nel Paragrafo 9.1.2 sono sufficienti per la maggior parte delle applicazioni elementari delle code prioritarie, come l'ordinamento, ma ci sono molte situazioni in cui sarebbero utili alcuni metodi in più, come dimostrato dagli scenari delineati nel seguito, relativi all'applicazione, già illustrata, che gestisce i passeggeri di una compagnia aerea in lista d'attesa per avere un posto su un volo.

- Un passeggero in lista d'attesa che abbia un'indole pessimista può stancarsi di aspettare e decidere di andarsene prima dell'imbarco, chiedendo di essere rimosso dalla lista. Di conseguenza, vorremmo poter rimuovere dalla coda prioritaria la voce associata a quel passeggero: l'operazione `removeMin` non è adatta, perché il passeggero che abbandona l'attesa non avrà necessariamente la priorità migliore. Ci serve una nuova operazione, `remove`, che elimini qualunque voce ricevuta come parametro.
- Un'altra passeggera è riuscita a ritrovare nella borsa la propria tessera di *frequent-flyer* e la mostra all'addetto alle prenotazioni, per cui la sua priorità deve essere modificata. Per realizzare questo cambio di priorità, vorremmo poter disporre di una nuova operazione, `replaceKey` che ci consenta di sostituire con una chiave nuova la chiave associata a una voce già presente nella coda prioritaria.
- Infine, un terzo passeggero in lista d'attesa ha notato che il suo nome è stato scritto sul biglietto in modo sbagliato e ha chiesto che venga corretto. Per questo cambiamento dobbiamo aggiornare i dati associati al passeggero, non la sua priorità, quindi ci serve una nuova operazione, `replaceValue`, che ci consenta, appunto, di sostituire con un nuovo valore il valore di una voce già presente nella coda prioritaria.

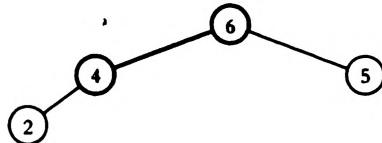
(a) A sequence of six numbers: 9, 7, 5, 2, 6, 4.



(b) A sequence of six numbers: 7, 6, 5, 2, 4, 9.



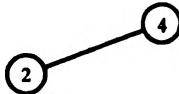
(c) A sequence of six numbers: 6, 4, 5, 2, 7, 9.



(d) A sequence of six numbers: 5, 4, 2, 6, 7, 9.



(e) A sequence of six numbers: 4, 2, 5, 6, 7, 9.



(f) A sequence of six numbers: 2, 4, 5, 6, 7, 9.



Figura 9.9: Fase 2 dell’algoritmo heap sort che opera sul posto. La porzione di ciascuna sequenza che è dedicata allo heap è rappresentata con lo sfondo grigio. L’albero binario implicitamente rappresentato da ciascuna sequenza è disegnato sulla destra, con il più recente percorso utilizzato dalla procedura di down-heap bubbling evidenziato.

Il tipo di dato astratto “coda prioritaria flessibile”

Gli scenari appena delineati suggeriscono la definizione di un nuovo tipo di dato astratto, la *coda prioritaria flessibile* (*adaptable priority queue*), che estende il comportamento dell’ADT “coda prioritaria”, aggiungendovi funzionalità. Nei Paragrafi 14.6.2 e 14.7.1, quando implementeremo alcuni algoritmi sui grafi, vedremo altre applicazioni delle code prioritarie flessibili.

Per implementare in modo efficiente i metodi `remove`, `replaceKey` e `replaceValue`, abbiamo bisogno di un meccanismo che rintracci una specifica voce all’interno della coda prioritaria, possibilmente senza fare una scansione lineare dell’intero contenitore. Nella definizione originaria dell’ADT coda prioritaria, un’invocazione di `insert(k, v)` restituisce all’utilizzatore un esemplare di tipo `Entry`. Per poter aggiornare una voce o rimuoverla all’interno del nostro nuovo ADT “coda prioritaria flessibile”, l’utilizzatore deve conservare quell’oggetto

di tipo `Entry` come “testimone” (*token*) da fornire come parametro per poter identificare la voce coinvolta. Dal punto di vista formale, l’ADT coda prioritaria flessibile contiene i metodi seguenti (oltre a quelli della coda prioritaria standard):

- `remove(e)`: Elimina la voce *e* dalla coda prioritaria.
- `replaceKey(e, k)`: Sostituisce con *k* la chiave della voce *e* già presente nella coda prioritaria.
- `replaceValue(e, v)`: Sostituisce con *v* il valore della voce *e* già presente nella coda prioritaria.

Se il parametro *e* fornito a uno di questi metodi non è valido (ad esempio, perché era già stato eliminato in precedenza dalla coda prioritaria), si verifica un errore.

9.5.1 Entry consapevoli della propria posizione

Per fare in modo che un esemplare di voce possa rappresentare la propria posizione all’interno di una coda prioritaria, estendiamo la classe `PQEntry` (originariamente definita all’interno della classe di base `AbstractPriorityQueue`) aggiungendo un terzo campo che contenga l’indice della voce all’interno dell’array usato per implementare lo heap, come si può vedere nella Figura 9.10 (un approccio simile al suggerimento che abbiamo dato nel Paragrafo 7.3.3 per implementare con un array il tipo di dato astratto “lista posizionale”). In questo modo le voci diventano “consapevoli della propria posizione” (*location-aware entry*).

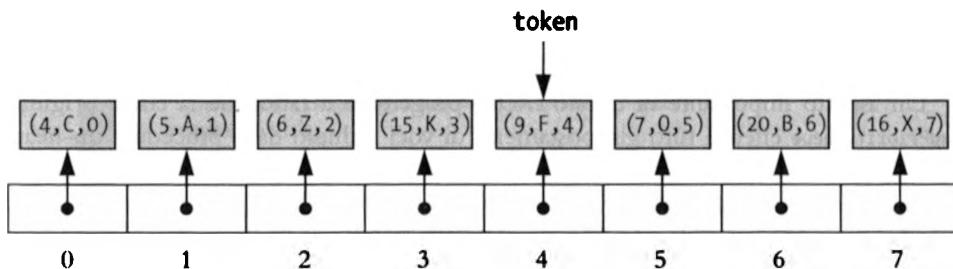


Figura 9.10: Rappresentazione di uno heap usando un array di voci consapevoli della propria posizione. Il terzo campo di ciascun esemplare di voce corrisponde all’indice di tale voce nell’array. La variabile *token* è riportata come esempio di un riferimento a una voce posto nell’ambito di visibilità dell’utilizzatore del contenitore.

Quando eseguiamo su questo heap le operazioni relative alla coda prioritaria, provocando spostamenti di alcune voci all’interno della struttura, dobbiamo aggiornare il terzo campo di ogni voce spostata, in modo che rappresenti la sua nuova posizione all’interno dell’array. Ad esempio, la Figura 9.11 mostra lo stato dello heap di Figura 9.10 dopo un’invocazione di `removeMin()`. L’operazione, agendo sullo heap, provoca la rimozione della voce che contiene la chiave minima, (4,C), mentre l’ultima voce, (16,X), si sposta temporaneamente dall’ultima posizione alla radice, per poi eseguire una fase di down-heap bubbling a partire proprio dalla radice. Durante questa procedura di down-heap, la voce (16,X) viene prima scambiata con il proprio figlio sinistro, (5,A), che si trova nella cella di indice 1 dell’array, poi con il

suo (nuovo) figlio destro, (9,F), che si trova nella cella di indice 4. Nella situazione finale, l'ultimo campo di tutte le voci coinvolte da spostamenti è stato modificato in modo da riflettere la loro nuova collocazione.

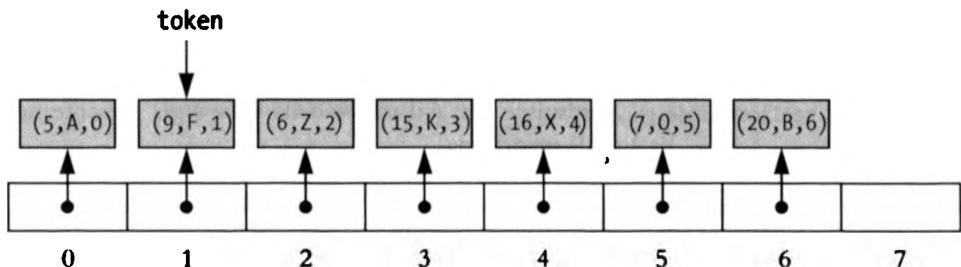


Figura 9.11: Risultato di un'invocazione di `removeMin()` sullo heap di Figura 9.10. La variabile `token` continua a fare riferimento alla stessa voce a cui puntava nella situazione di partenza, ma la posizione di tale voce nell'array è cambiata, come si può notare osservando il valore del suo terzo campo.

9.5.2 Implementare una coda prioritaria flessibile

Il Codice 9.12 e 9.13 presenta un'implementazione in Java di una coda prioritaria flessibile, definita come sottoclasse della classe `HeapPriorityQueue` vista nel Paragrafo 9.3.2. Iniziamo definendo una classe annidata, `AdaptablePQEntry` (nelle righe che vanno da 5 a 15), che estende la classe `PQEntry` ereditata, aggiungendovi il campo `index`. Il metodo ereditato `insert` viene sovrascritto, in modo che crei e inizializzi un esemplare della classe `AdaptablePQEntry` e non della classe originaria `PQEntry`.

Un aspetto importante di questo nostro progetto è il fatto che la classe originaria `HeapPriorityQueue` effettui tutti gli spostamenti di voci richiesti dalle procedure `up-heap` e `down-heap` bubbling usando solamente il metodo `protected swap`: è sufficiente, quindi, che la classe `AdaptablePriorityQueue` sovrascriva quel metodo ausiliario in modo che aggiorni gli indici memorizzati nelle voci consapevoli della propria posizione nel momento in cui vengono spostate (come già detto).

Quando una voce viene fornita come parametro ai metodi `remove`, `replaceKey` e `replaceValue`, sfruttiamo il fatto che il suo nuovo campo `index` identifichi la cella in cui si trova tale voce all'interno dell'array che implementa lo heap (una proprietà, fra l'altro, che può essere verificata facilmente). Quando viene sostituita la chiave di una voce già presente nello heap, tale nuova chiave può violare la proprietà di ordinamento dello heap, essendo troppo piccola o troppo grande. Definiamo, quindi, un nuovo metodo ausiliario, `bubble`, che determina se sia necessaria un'esecuzione della procedura `up-heap` o `down-heap`. Quando si soddisfa la richiesta di eliminazione di una determinata voce, la si sostituisce con l'ultima voce presente nello heap (cioè con la voce memorizzata nell'ultima posizione dello heap, in modo da eliminare tale ultima posizione e preservare la proprietà di completezza dell'albero binario) e si esegue il metodo `bubble`, perché la voce che è stata spostata può, di nuovo, avere una chiave troppo piccola o troppo grande per la nuova posizione che occupa.

Codice 9.12: Un'implementazione di coda prioritaria flessibile (che prosegue nel Codice 9.13). Questa classe estende la classe HeapPriorityQueue definita nel Codice 9.8 e 9.9.

```

1  /** Una coda prioritaria flessibile che usa uno heap realizzato con un array. */
2  public class HeapAdaptablePriorityQueue<K,V> extends HeapPriorityQueue<K,V>
3      implements AdaptablePriorityQueue<K,V> {
4
5      //----- classe AdaptablePQEntry annidata -----
6      /** Estensione della classe PQEntry per aggiungere informazioni sulla posizione. */
7      protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
8          private int index; // indice della voce nell'array che realizza lo heap
9          public AdaptablePQEntry(K key, V value, int j) {
10              super(key, value); // memorizza chiave e valore
11              index = j;
12          }
13          public int getIndex() { return index; }
14          public void setIndex(int j) { index = j; }
15      } //---- fine della classe AdaptablePQEntry annidata -----
16
17      /** Crea una coda prioritaria flessibile vuota che usa l'ordine naturale. */
18      public HeapAdaptablePriorityQueue() { super(); }
19      /** Crea una coda prioritaria flessibile vuota che usa il comparatore fornito. */
20      public HeapAdaptablePriorityQueue(Comparator<K> c) { super(c); }
21
22      // metodi ausiliari protected
23      /** Determina se una voce è valida, controllando la sua posizione. */
24      protected AdaptablePQEntry<K,V> validate(Entry<K,V> entry)
25          throws IllegalArgumentException {
26          if (!(entry instanceof AdaptablePQEntry))
27              throw new IllegalArgumentException("Invalid entry");
28          AdaptablePQEntry<K,V> locator = (AdaptablePQEntry<K,V>) entry; // cast sicuro
29          int j = locator.getIndex();
30          if (j >= heap.size() || heap.get(j) != locator)
31              throw new IllegalArgumentException("Invalid entry");
32          return locator;
33      }
34
35      /** Scambia tra loro le voci delle celle i e j dell'array. */
36      protected void swap(int i, int j) {
37          super.swap(i, j); // esegue lo scambio
38          ((AdaptablePQEntry<K,V>) heap.get(i)).setIndex(i); // sistema l'indice
39          ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(j); // sistema l'indice
40      }

```

Codice 9.13: Un'implementazione di coda prioritaria flessibile (continua dal Codice 9.13).

```

41      /** Ripristina la proprietà di ordinamento spostando la voce j in alto/basso. */
42      protected void bubble(int j) {
43          if (j > 0 && compare(heap.get(j), heap.get(parent(j))) < 0)
44              upheap(j);
45          else
46              downheap(j); // però può darsi che non sia necessario alcuno spostamento
47      }
48
49      /** Inserisce una coppia chiave-valore e restituisce la voce creata. */
50      public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
51          checkKey(key); // può lanciare eccezione
52          Entry<K,V> newest = new AdaptablePQEntry<>(key, value, heap.size());

```

```

53     heap.add(newest);           // aggiunge alla fine dell'array
54     upheap(heap.size() - 1);   // esegue up-heap bubbling sulla voce aggiunta
55     return newest;
56 }
57
58 /** Elimina dalla coda prioritaria la voce ricevuta. */
59 public void remove(Entry<K,V> entry) throws IllegalArgumentException {
60     AdaptablePQEntry<K,V> locator = validate(entry);
61     int j = locator.getIndex();
62     if (j == heap.size() - 1);      // la voce si trova nell'ultima posizione
63         heap.remove(heap.size() - 1); // quindi basta eliminarla
64     else {
65         swap(j, heap.size() - 1); // scambia con la voce in ultima posizione
66         heap.remove(heap.size() - 1); // poi elimina la voce spostata alla fine
67         bubble(j);                // e sistema l'altra voce scambiata
68     }
69 }
70
71 /** Sostituisce la chiave di una voce. */
72 public void replaceKey(Entry<K,V> entry, K key)
73             throws IllegalArgumentException {
74     AdaptablePQEntry<K,V> locator = validate(entry);
75     checkKey(key);           // può lanciare eccezione
76     locator.setKey(key);    // metodo ereditato da PQEntry
77     bubble(locator.getIndex()); // con la nuova chiave, forse serve un aggiustamento
78 }
79
80 /** Sostituisce il valore di una voce. */
81 public void replaceValue(Entry<K,V> entry, V value)
82             throws IllegalArgumentException {
83     AdaptablePQEntry<K,V> locator = validate(entry);
84     locator.setValue(value); // metodo ereditato da PQEntry
85 }
86 }

```

Prestazioni delle implementazioni di coda prioritaria flessibile

La Tabella 9.5 riassume le prestazioni di una coda prioritaria flessibile implementata mediante una struttura a heap con entità consapevoli della propria posizione. La nuova classe garantisce le stesse prestazioni asintotiche e la stessa occupazione di spazio in memoria della versione originaria, non flessibile, con prestazioni logaritmiche per i nuovi metodi `remove` e `replaceKey` basati su entità *location-aware* e, infine, prestazioni tempo-costanti per il nuovo metodo `replaceValue`.

Tabella 9.5: Tempi d'esecuzione dei metodi di una coda prioritaria flessibile di dimensione n , realizzata mediante heap memorizzato in un array. Lo spazio utilizzato è $O(n)$.

Metodo	Tempo d'esecuzione
<code>size</code> , <code>isEmpty</code> , <code>min</code>	$O(1)$
<code>insert</code>	$O(\log n)$
<code>remove</code>	$O(\log n)$
<code>removeMin</code>	$O(\log n)$
<code>replaceKey</code>	$O(\log n)$
<code>replaceValue</code>	$O(1)$

9.6 Esercizi

Riepilogo e approfondimento

- R-9.1 Usando il metodo `removeMin`, quanto tempo serve per rimuovere gli elementi minori, in numero uguale a $\lceil \log n \rceil$, da uno heap contenente n elementi?
- R-9.2 Se in un albero binario T a ogni posizione p viene assegnata una chiave uguale al proprio rango nell'attraversamento in pre-ordine, a quali condizioni T è uno heap?
- R-9.3 Eseguendo questa sequenza di operazioni su una coda prioritaria, cosa restituisce ciascuna invocazione di `removeMin?` `insert(5, A)`, `insert(4, B)`, `insert(7, F)`, `insert(1, D)`, `removeMin()`, `insert(3, J)`, `insert(6, L)`, `removeMin()`, `removeMin()`, `insert(8, G)`, `removeMin()`, `insert(2, H)`, `removeMin()`, `removeMin()`.
- R-9.4 Un aeroporto sta sviluppando un simulatore computerizzato del controllo del traffico aereo, in grado di gestire eventi come decolli e atterraggi. A ciascun evento è associato un istante di tempo (*time stamp*) che indica il momento in cui l'evento accadrà. Il programma di simulazione deve eseguire in modo efficiente le due seguenti operazioni fondamentali:
- Inserire un evento associato a un determinato istante di tempo (nel futuro).
 - Estrarre l'evento associato all'istante di tempo minimo (determinando, così, il prossimo evento che dovrà accadere).

Quale struttura dati si dovrebbe usare per risolvere il problema? Perché?

- R-9.5 Il metodo `min` della classe `UnsortedPriorityQueue` viene eseguito in un tempo $O(n)$, come visto nella Tabella 9.2. Fornire una semplice modifica da apportare alla classe in modo che il metodo `min` diventi $O(1)$. Illustrare anche eventuali modifiche che si rendano necessarie ad altri metodi della classe.
- R-9.6 È possibile adattare la soluzione fornita all'esercizio precedente in modo da rendere $O(1)$ anche il tempo d'esecuzione del metodo `removeMin` nella classe `UnsortedPriorityQueue`? Spiegare adeguatamente la risposta.
- R-9.7 Illustrare l'esecuzione dell'algoritmo di ordinamento per selezione con questa sequenza iniziale: (22, 15, 36, 44, 10, 3, 9, 13, 29, 25).
- R-9.8 Illustrare l'esecuzione dell'algoritmo di ordinamento per inserimento con la sequenza iniziale dell'esercizio precedente.
- R-9.9 Fornire un esempio di sequenza di n elementi che costituisca caso peggiore per l'ordinamento per inserimento e dimostrare che la sua esecuzione su tale sequenza richiede un tempo $\Omega(n^2)$.
- R-9.10 In quali posizioni di uno heap potrebbe essere memorizzata la terza chiave più piccola?
- R-9.11 In quali posizioni di uno heap potrebbe essere memorizzata la chiave massima?
- R-9.12 Come si potrebbe utilizzare una normale coda prioritaria (orientata alla priorità minima) per gestire una situazione in cui si hanno chiavi numeriche e si ha bisogno di una coda prioritaria orientata alla priorità massima?
- R-9.13 Illustrare l'esecuzione dell'algoritmo di ordinamento sul posto mediante heap (*in-place heap sort*) con questa sequenza iniziale: (2, 5, 16, 4, 10, 23, 39, 18, 26, 15).

- R-9.14 Sia T un albero binario completo tale che ciascuna posizione p memorizzi una voce avente chiave $f(p)$, la funzione di numerazione per livelli descritta nel Paragrafo 8.3.2. T è uno heap? Perché?
- R-9.15 Spiegare perché la descrizione della procedura di *down-heap bubbling* non prende in considerazione il caso in cui la posizione p ha il figlio destro senza avere il figlio sinistro.
- R-9.16 Esiste uno heap H che contiene sette voci aventi chiavi tutte distinte, tale che l'attraversamento in pre-ordine di H visiti le voci di H in ordine di chiave crescente o decrescente? E se si usa l'attraversamento in ordine simmetrico? E in post-ordine? Per ogni risposta affermativa fornire un esempio; per ogni risposta negativa, spiegare il motivo.
- R-9.17 Sia H uno heap che contiene 15 voci e usa la rappresentazione di albero binario completo mediante array. Qual è la sequenza di indici che corrisponde all'ordine in cui vengono visitati gli elementi di H durante il suo attraversamento in pre-ordine? E in ordine simmetrico? E in post-ordine?
- R-9.18 Dimostrare che la somma $\sum_{i=1}^n \log i$, che compare nell'analisi delle prestazioni di *heap sort*, è $\Omega(n \log n)$.
- R-9.19 Bill afferma che l'attraversamento in pre-ordine di uno heap elenca le sue chiavi in ordine non decrescente. Disegnare un esempio di heap che dimostri che sbaglia.
- R-9.20 Hillary afferma che l'attraversamento in post-ordine di uno heap elenca le sue chiavi in ordine non crescente. Disegnare un esempio di heap che dimostri che sbaglia.
- R-9.21 Illustrare tutti i passi eseguiti da una coda prioritaria flessibile realizzata mediante lo heap della Figura 9.1 che esegue l'invocazione `remove(e)` quando e fa riferimento all'entità (16,X).
- R-9.22 Illustrare tutti i passi eseguiti da una coda prioritaria flessibile realizzata mediante lo heap della Figura 9.1 che esegue l'invocazione `replaceKey(e, 18)` quando e fa riferimento all'entità (5,A).
- R-9.23 Disegnare un esempio di heap le cui chiavi siano tutti i numeri dispari da 1 a 59 (senza ripetizioni), in modo che l'inserimento di una voce avente chiave 32 provochi, mediante applicazione della procedura *up-heap bubbling*, la sua risalita fino a un figlio della radice (nel quale andrà, quindi, a posizionarsi la chiave 32).
- R-9.24 Descrivere una sequenza di n inserimenti in uno heap che richieda un tempo d'esecuzione $\Omega(n \log n)$.

Creatività

- C-9.25 Spiegare come si possa implementare l'ADT "pila" usando soltanto una coda prioritaria e una variabile di esemplare di tipo numerico intero.
- C-9.26 Spiegare come si possa implementare l'ADT "coda FIFO" usando soltanto una coda prioritaria e una variabile di esemplare di tipo numerico intero.
- C-9.27 Il Professor Inutile suggerisce, per il problema precedente, la soluzione seguente. Ogni volta che un valore viene inserito nella coda, gli viene associata una chiave uguale alla dimensione attuale della coda stessa, per poi inserire tale coppia chiave-valore nella coda prioritaria. Si ottiene così una semantica FIFO? Dimostrare che sia così, oppure fornire un controesempio.

- C-9.28 Implementare nuovamente in Java la classe `SortedPriorityQueue` usando un array e garantendo che il metodo `removeMin` mantenga prestazioni $O(1)$.
- C-9.29 Fornire un'implementazione del metodo `upheap` della classe `HeapPriorityQueue` che usi la ricorsione e non abbia cicli.
- C-9.30 Fornire un'implementazione del metodo `downheap` della classe `HeapPriorityQueue` che usi la ricorsione e non abbia cicli.
- C-9.31 Si ipotizzi di usare una rappresentazione concatenata per un albero binario completo T , con un riferimento aggiuntivo che punti al suo ultimo nodo. Dimostrare come si possa aggiornare tale riferimento dopo un'operazione `insert` o `remove` in un tempo $O(\log n)$, essendo n il numero di nodi di T . Accertarsi di aver considerato tutti i casi possibili, illustrati nella Figura 9.12.

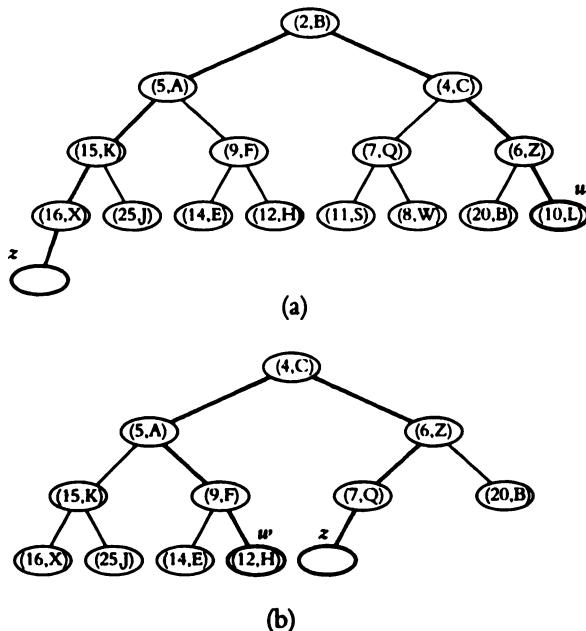


Figura 9.12: Due casi di aggiornamento dell'ultimo nodo in un albero binario completo dopo operazioni di inserimento o rimozione. Il nodo *w* è l'ultimo nodo dopo un'operazione di inserimento o prima di un'operazione di rimozione. Il nodo *z* è l'ultimo nodo prima di un'operazione di inserimento o dopo un'operazione di rimozione.

- C-9.32 Quando si usa una rappresentazione di heap mediante albero concatenato, un metodo alternativo per trovare l'ultimo nodo durante un'operazione di inserimento in uno heap T consiste nella memorizzazione, in ogni foglia di T , di un riferimento alla foglia immediatamente più a destra (passando al primo nodo del livello successivo nel caso della foglia che si trova nella posizione più a destra possibile in un livello). Spiegare come si possano aggiornare tali riferimenti in un tempo $O(1)$ durante l'esecuzione di ciascuna operazione dell'ADT coda prioritaria.
- C-9.33 Possiamo rappresentare un percorso che vada dalla radice a un determinato nodo di un albero binario mediante una stringa binaria, dove 0 significa "scendi nel figlio

sinistro” e 1 significa “scendi nel figlio destro”. Ad esempio, il percorso che va dalla radice al nodo che contiene $(8, W)$ nello heap della Figura 9.12a si può rappresentare con “101”. Progettare un algoritmo che venga eseguito in un tempo $O(\log n)$ e che trovi l’ultimo nodo di un albero binario completo avente n nodi, sfruttando la rappresentazione appena descritta. Spiegare come si possa utilizzare questo algoritmo nell’implementazione di un albero binario completo realizzato con una struttura concatenata che non memorizza un riferimento esplicito all’ultimo nodo.

- C-9.34 Dato uno heap H e una chiave k , descrivere un algoritmo che trovi tutte le entità di H che hanno una chiave non maggiore di k . Ad esempio, dato lo heap della Figura 9.12a e la chiave $k = 7$, l’algoritmo deve trovare le entità che hanno come chiave i numeri 2, 4, 5, 6 e 7 (non necessariamente in questo ordine). L’algoritmo deve essere eseguito in un tempo proporzionale al numero di entità restituite e non deve modificare lo heap.

C-9.35 Dimostrare le prestazioni temporali riportate nella Tabella 9.5.

- C-9.36 Dimostrare in modo diverso le prestazioni temporali della costruzione *bottom-up* di uno heap, mostrando che la somma seguente è $O(1)$ per qualsiasi numero intero positivo h :

$$\sum_{i=1}^h (i/2^i).$$

- C-9.37 Ipotizzando che due alberi binari, T_1 e T_2 , contengano entità che soddisfano la proprietà di ordinamento di uno heap (ma non necessariamente anche la proprietà di completezza degli alberi binari), descrivere un metodo che li combini in un unico albero binario T , i cui nodi contengano l’unione delle entità di T_1 e T_2 , rispettando la proprietà di ordinamento di uno heap. L’algoritmo deve richiedere un tempo d’esecuzione $O(h_1 + h_2)$, dove h_1 e h_2 sono, rispettivamente, l’altezza di T_1 e di T_2 .

- C-9.38 La linea aerea Tamarindo Airlines vuole offrire un omaggio ai suoi passeggeri migliori, in numero di $\log n$, sulla base della quantità di miglia volate, dove n è il numero complessivo di passeggeri della compagnia. L’algoritmo che viene attualmente utilizzato ordina i passeggeri in base al numero di miglia volate e, poi, scandisce la lista per scegliere i migliori, in un tempo complessivo $O(n \log n)$. Descrivere un algoritmo che identifichi i passeggeri da premiare in un tempo $O(n)$.

- C-9.39 Spiegare come, usando uno heap orientato alla chiave massima, si possano trovare i k elementi maggiori di un contenitore non ordinato di dimensione n in un tempo $O(n + k \log n)$.

- C-9.40 Spiegare come, usando uno spazio aggiuntivo $O(k)$, si possano trovare i k elementi maggiori di un contenitore non ordinato di dimensione n in un tempo $O(n \log k)$.

- C-9.41 Scrivere un comparatore per numeri interi non negativi che determini il risultato del confronto sulla base del numero di cifre 1 presenti nella rappresentazione binaria di ciascun numero, in modo che sia $i < j$ se e solo se il numero di cifre 1 presenti nella rappresentazione binaria di i è minore del numero di cifre 1 presenti nella rappresentazione binaria di j .

- C-9.42 Implementare l’algoritmo `binarySearch` (visto nel Paragrafo 5.1.3) per un array i cui elementi siano del tipo generico `E`, usando un oggetto di tipo `Comparator`.

- C-9.43 Data una classe, `MinPriorityQueue`, che implementa il tipo di dato astratto “coda prioritaria orientata alla chiave minima”, scrivere un’implementazione della classe `MaxPriorityQueue` che usi un suo adattamento per realizzare l’analoga astrazione orientata alla chiave massima, con i metodi `insert`, `max` e `removeMax`. L’implementazione proposta non deve fare nessuna ipotesi sui dettagli interni alla classe `MinPriorityQueue` originaria, né sul tipo di chiavi che possono essere utilizzate.
- C-9.44 Descrivere una versione “sul posto” dell’algoritmo di ordinamento per selezione di un array che usi soltanto uno spazio di memoria $O(1)$ per le proprie variabili, oltre all’array.
- C-9.45 Ipotizzando che, in un problema di ordinamento, i dati vengano forniti in un array A , descrivere come si possa implementare l’algoritmo di ordinamento per inserimento usando soltanto l’array A e al massimo sei variabili aggiuntive (di tipi fondamentali).
- C-9.46 Fornire una descrizione alternativa dell’algoritmo *heap sort* che opera sul posto usando una coda prioritaria standard, orientata alla chiave minima (invece di una orientata alla chiave massima).
- C-9.47 Un gruppo di bambini vuole fare una partita a un gioco, chiamato *UnMonopoly*, dove, a turno, il giocatore che ha più denaro deve dare la metà del proprio denaro al giocatore che ne ha di meno. Quale struttura dati si può usare per simulare in modo efficiente questo gioco? Perché?
- C-9.48 Un sistema di elaborazione *online* per il commercio di azioni deve elaborare ordini nella forma “buy 100 shares at \$ x each” oppure “sell 100 shares at \$ y each” (cioè “compra/vendi 100 azioni al prezzo unitario di \$ x /\$ y ”). Un ordine di acquisto al prezzo \$ x può essere elaborato soltanto se esiste un ordine di vendita al prezzo \$ y , con $y \leq x$. Analogamente, un ordine di vendita al prezzo \$ y può essere elaborato soltanto se esiste un ordine di acquisto al prezzo \$ x , con $y \leq x$. Se un ordine di acquisto o di vendita viene inserito ma non può essere elaborato, rimane in attesa per un ordine futuro che consenta la sua elaborazione. Descrivere uno schema che consenta di inserire un ordine di acquisto o di vendita in un tempo $O(\log n)$, indipendentemente dal fatto che possa essere elaborato immediatamente oppure no.
- C-9.49 Estendere la soluzione dell’esercizio precedente in modo che si possa aggiornare il prezzo di acquisto o di vendita per ordini che non siano ancora stati elaborati.

Progettazione

- P-9.50 Implementare l’algoritmo *heap sort* che opera sul posto. Confrontare sperimentalmente il suo tempo d’esecuzione con quello dell’algoritmo *heap sort* standard, che non opera sul posto.
- P-9.51 Usare uno degli approcci visti negli Esercizi C-9.39 e C-9.40 per implementare in modo alternativo il metodo `getFavorites` della classe `FavoritesListMTF` vista nel Paragrafo 7.7.2, garantendo che i risultati vengano generati a partire dal più grande per arrivare al più piccolo.
- P-9.52 Sviluppare un’implementazione, in Java, di una coda prioritaria flessibile che sia basata su una lista non ordinata e usi entità *location-aware*.
- P-9.53 Scrivere un programma grafico (o un *applet* Java) che animi uno heap, consentendo l’esecuzione di tutte le operazioni della coda prioritaria e visualizzando gli scambi che avvengono durante le procedure *up-heap bubbling* e *down-heap bubbling*, eventualmente visualizzando anche la costruzione *bottom-up*.

- P-9.54 Scrivere un programma che elabori una sequenza di ordini di acquisto e vendita di azioni, come descritto nell'Esercizio C-9.48.
- P-9.55 Una delle principali applicazioni delle code prioritarie riguarda i sistemi operativi, per decidere quali processi (*job*) eseguire sulla CPU (problema dello *scheduling* dei processi). In questo progetto darete vita a un programma che simula le decisioni da prendere per eseguire processi sulla CPU di un calcolatore. Il programma deve eseguire continuamente un ciclo, ciascuna iterazione del quale corrisponde a un *intervallo temporale* (*time slice*) della CPU. A ogni processo viene assegnata una priorità, che è un numero intero compreso tra -20 (priorità massima) e 19 (priorità minima), estremi inclusi. Tra tutti i processi che attendono di ricevere un intervallo temporale dedicato alla propria esecuzione, la CPU deve operare su quello avente la priorità massima. In questa simulazione, ogni processo avrà anche un valore di *durata* o lunghezza (*length*), un numero intero positivo minore di 101, che indica il numero di intervalli temporali necessari per portarlo a termine. Per semplicità, si può ipotizzare che un processo non possa essere interrotto: una volta che è stato messo in esecuzione nella CPU, un processo viene eseguito per un numero di intervalli di tempo uguale alla sua lunghezza. Il simulatore deve visualizzare il nome del processo che, in ciascun intervallo di tempo, si trova in esecuzione nella CPU, e deve elaborare una sequenza di comandi, uno per ciascun intervallo di tempo, aventi la forma “add job *name* with length *n* and priority *p*” (cioè “aggiungi il processo *name* con lunghezza *n* e priorità *p*”) oppure “no new job this slice” (cioè “nessun nuovo processo in questo intervallo di tempo”).
- P-9.56 Sia S un insieme di n punti in un piano aventi numeri interi tutti distinti come coordinate x e y . Sia T un albero binario completo che memorizza i punti di S come suoi nodi esterni, in modo che i punti siano ordinati da sinistra a destra per coordinata x crescente. Per ogni nodo v in T , indichiamo con $S(v)$ il sottoinsieme di S costituito dai punti memorizzati nel sottoalbero di T avente radice v . Per la radice r di T , chiamiamo $\text{top}(r)$ il punto avente coordinata y massima tra quelli appartenenti a $S(r) = S$. Per ogni altro nodo v , chiamiamo $\text{top}(v)$ il punto avente coordinata y massima tra quelli appartenenti a $S(v)$, che non sia però anche la coordinata y massima in $S(u)$, dove u è il genitore di v in T (che, a questo punto, esiste senza dubbio). Questa assegnazione di etichette trasforma T in un *albero di ricerca con priorità* (*priority search tree*). Descrivere un algoritmo che in un tempo lineare trasformi T in un albero di ricerca con priorità, poi implementarlo in Java.

Note

Il testo di Knuth dedicato a ordinamento e ricerca [61] descrive le motivazioni e la storia degli algoritmi *selection sort*, *insertion sort* e *heap sort*. L'algoritmo *heap sort* è dovuto a Williams [95] e l'algoritmo di costruzione di uno heap in un tempo lineare è di Floyd [35]. Nei lavori di Bentley [14], Carlsson [21], Gonnet e Munro [39], McDiarmid e Reed [69] e Schaffer e Sedgewick [82] sono riportati ulteriori algoritmi e analisi di varianti di heap e di *heap sort*.

10

Mappe, tabelle hash e skip list

10.1 Mappe

Una **mappa** (*map*) è un tipo di dato astratto progettato per agire in modo efficiente nella memorizzazione e nel recupero di valori sulla base di una **chiave di ricerca** (*search key*) che li identifica in modo univoco. Nello specifico, una mappa memorizza coppie chiave-valore (k, v), che chiamiamo **voci** (*entry*), dove k è la chiave e v è il valore che le corrisponde. Il fatto che le chiavi siano tutte distinte è un requisito e l'associazione tra chiavi e valori si chiama anche mappatura, *mapping*. La Figura 10.1 illustra il concetto di mappa usando la metafora dell'archivio a cartelle o schede. Come metafora più moderna, pensate al Web come a una mappa, le cui voci sono le pagine web: la chiave di una pagina è il suo URL (*uniform resource locator*, una stringa come <http://datastructures.net/>) e il suo valore è il contenuto della pagina stessa.

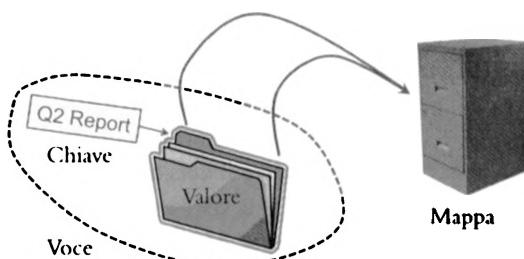


Figura 10.1: Metafora del concetto di mappa come tipo di dato astratto. Le chiavi (etichette) vengono assegnate ai valori (le cartelle di documenti) dall'utente. Le voci che ne risultano (le cartelle con etichetta) vengono inserite nella mappa (l'archivio a cassetti). Le chiavi possono, poi, essere utilizzate per recuperare o eliminare i valori.

Le mappe sono note anche con il nome di *array associativi* (*associative array*), perché la chiave di una voce ha in qualche modo la funzione di un indice, all'interno della mappa, che aiuta la mappa stessa a localizzare in modo efficiente la voce associata. Tuttavia, diversamente dagli array standard, non è necessario che, in una mappa, la chiave sia numerica, e non identifica direttamente una posizione all'interno della struttura. Tra le comuni applicazioni di mappe, citiamo le seguenti.

- Il sistema informativo di un'università si basa su una qualche forma di identificativo univoco degli studenti (ID), come chiave per associare uno studente all'insieme dei dati (*record*) che lo riguardano (come il nome, l'indirizzo e i voti ottenuti negli esami), che ha il ruolo del valore.
- Il sistema dei nomi di dominio (DNS, *domain name system*) di Internet costituisce una mappatura tra i nomi degli *host*, come `www.wiley.com`, e gli indirizzi IP (*Internet Protocol*), come `208.215.179.146`.
- Il sito di un *social media* usa tipicamente un “nome utente” (*username*), non numerico, come chiave che possa essere messa rapidamente in corrispondenza con le informazioni associate a quel particolare utente.
- L'insieme dei clienti di un'azienda può essere memorizzata in una mappa, usando come chiave un “codice cliente” o un'altra informazione univoca (che chiamiamo genericamente ID) e come valore le informazioni (*record*) relative al cliente. La mappa consentirà a un agente rappresentante dell'azienda di accedere rapidamente alle informazioni relative al cliente, data la chiave.
- Un sistema grafico per computer mette in corrispondenza, mediante una mappa, i nomi dei colori, come ‘turquoise’, con le terne di numeri che ne descrivono la rappresentazione RGB, come `(64, 224, 208)`.

10.1.1 La mappa come tipo di dato astratto

Dato che una mappa memorizza una raccolta di oggetti, la si dovrebbe considerare una collezione di coppie chiave-valore. Come tipo di dato astratto, una *mappa* (*map*) M mette a disposizione i seguenti metodi:

- `size()`:** Restituisce il numero di voci presenti nella mappa M .
- `isEmpty()`:** Restituisce `true` se e solo se la mappa M è vuota.
- `get(k)`:** Restituisce il valore v associato alla chiave k , se nella mappa esiste la voce (k, v) , altrimenti restituisce `null`.
- `put(k, v)`:** Se M non contiene una voce avente chiave uguale a k , aggiunge la voce (k, v) a M e restituisce `null`; altrimenti, sostituisce v al valore presente nella voce avente chiave uguale a k e restituisce il vecchio valore, che è stato sostituito.
- `remove(k)`:** Elimina da M la voce avente chiave uguale a k e ne restituisce il valore; se M non contiene una tale voce, restituisce `null`.
- `keySet()`:** Restituisce un contenitore iterabile contenente tutte le *chiavi* presenti nelle voci memorizzate in M .

values(): Restituisce un contenitore iterabile contenente tutti i *valori* presenti nelle voci memorizzate in *M* (con eventuali duplicati se, nella mappa, più chiavi sono associate a uno stesso valore).

entrySet(): Restituisce un contenitore iterabile contenente tutte le voci di tipo chiave-valore memorizzate in *M*.

Mappe nel pacchetto java.util

La nostra definizione di mappa come ADT è una versione semplificata dell'interfaccia `java.util.Map` e per gli elementi del contenitore restituito dal metodo `entrySet` usiamo l'interfaccia per oggetti composti `Entry` già definita nel Paragrafo 9.2.1 (mentre `java.util.Map` usa l'interfaccia annidata `java.util.Map.Entry`).

Osserviamo che ciascuna delle operazioni `get(k)`, `put(k, v)` e `remove(k)` restituisce il valore associato alla chiave *k*, se la mappa contiene una tale voce, altrimenti restituiscono `null`. Questa scelta introduce una possibile ambiguità in un'applicazione in cui `null` è consentito come normale valore associato a una chiave: infatti, se nella mappa è presente una voce `(k, null)`, l'operazione `get(k)` restituirà `null`, non perché non ha trovato la chiave, ma perché l'ha trovata e ha restituito il valore associato.

Alcune implementazioni dell'interfaccia `java.util.Map` impediscono esplicitamente l'uso di `null` come valore (e anche come chiave, peraltro). Tuttavia, per risolvere l'ambiguità nei casi in cui il valore `null` è consentito dall'applicazione, l'interfaccia contiene anche un metodo, `containsKey(k)`, che restituisce un valore booleano, `true` se e solo se la chiave *k* è presente nella mappa (lasciamo come esercizio l'implementazione di questo metodo).

Esempio 10.1: La tabella seguente mostra una sequenza di operazioni e i loro effetti su una mappa, inizialmente vuota, che usa numeri interi come chiavi e singoli caratteri come valori.

Metodo	Valore restituito	Contenuto della mappa
<code>isEmpty()</code>	<code>true</code>	<code>{}</code>
<code>put(5,A)</code>	<code>null</code>	<code>{(5,A)}</code>
<code>put(7,B)</code>	<code>null</code>	<code>{(5,A), (7,B)}</code>
<code>put(2,C)</code>	<code>null</code>	<code>{(5,A), (7,B), (2,C)}</code>
<code>put(8,D)</code>	<code>null</code>	<code>{(5,A), (7,B), (2,C), (8,D)}</code>
<code>put(2,E)</code>	<code>C</code>	<code>{(5,A), (7,B), (2,E), (8,D)}</code>
<code>get(7)</code>	<code>B</code>	<code>{(5,A), (7,B), (2,E), (8,D)}</code>
<code>get(4)</code>	<code>null</code>	<code>{(5,A), (7,B), (2,E), (8,D)}</code>
<code>get(2)</code>	<code>E</code>	<code>{(5,A), (7,B), (2,E), (8,D)}</code>
<code>size()</code>	<code>4</code>	<code>{(5,A), (7,B), (2,E), (8,D)}</code>
<code>remove(5)</code>	<code>A</code>	<code>{(7,B), (2,E), (8,D)}</code>
<code>remove(2)</code>	<code>E</code>	<code>{(7,B), (8,D)}</code>
<code>get(2)</code>	<code>null</code>	<code>{(7,B), (8,D)}</code>
<code>remove(2)</code>	<code>null</code>	<code>{(7,B), (8,D)}</code>
<code>isEmpty()</code>	<code>false</code>	<code>{(7,B), (8,D)}</code>
<code>entrySet()</code>	<code>{(7,B), (8,D)}</code>	<code>{(7,B), (8,D)}</code>
<code>keySet()</code>	<code>{7,8}</code>	<code>{(7,B), (8,D)}</code>
<code>values()</code>	<code>{B,D}</code>	<code>{(7,B), (8,D)}</code>

Un'interfaccia Java per il tipo di dato astratto "mappa"

Il Codice 10.1 riporta la nostra versione di interfaccia Java per l'ADT mappa: usa l'infrastruttura per la programmazione generica (vista nel Paragrafo 2.5.2), con *K* che individua il tipo di dato delle chiavi e *V* che, analogamente, individua il tipo di dato dei valori.

Codice 10.1: Interfaccia Java per la nostra versione semplificata dell'ADT mappa.

```

1 public interface Map<K,V> {
2     int size();
3     boolean isEmpty();
4     V get(K key);
5     V put(K key, V value);
6     V remove(K key);
7     Iterable<K> keySet();
8     Iterable<V> values();
9     Iterable<Entry<K,V>> entrySet();
10 }
```

10.1.2 Applicazione: frequenza delle parole in un testo

Come caso di studio per l'utilizzo di una mappa, consideriamo il problema di contare il numero di occorrenze di ciascuna parola in un documento, un problema standard quando se ne esegue un'analisi statistica, ad esempio per catalogare messaggi di posta elettronica o articoli di giornale. Una mappa è una struttura dati ideale per risolvere questo problema, usando le parole come chiavi e i relativi conteggi come valori. Nel Codice 10.2 presentiamo questa applicazione.

Partiamo da una mappa vuota, mettendo in corrispondenza le parole con i loro conteggi (detti anche, in questo contesto, "frequenze"), utilizzando la classe *ChainHashMap* di cui parleremo nel Paragrafo 10.2.4. Per prima cosa effettuiamo una scansione del testo in ingresso, considerando come parole le sequenze di caratteri alfabetici adiacenti, che convertiamo in minuscolo. Per ogni parola trovata, cerchiamo di recuperare dalla mappa il suo conteggio attuale usando il metodo *get*, osservando che, ovviamente, a una parola non ancora vista verrà attribuito il conteggio zero. Poi, (re)impostiamo il conteggio associato alla parola in modo che sia uguale a un'unità in più del suo valore precedente, per riflettere la nuova occorrenza della parola. Dopo aver elaborato in questo modo l'intero documento, eseguiamo un ciclo sulle voci presenti nel contenitore restituito dall'invocazione del metodo *entrySet()* della mappa per determinare quale parola abbia la frequenza massima.

Codice 10.2: Un programma che conta le occorrenze delle parole in un documento, visualizzando la parola più frequente. Il documento viene acquisito usando la classe *Scanner*, nella quale modifichiamo il delimitatore che separa i *token* in modo che usi qualunque carattere che non sia una lettera dell'alfabeto. Inoltre, convertiamo tutte le parole in minuscolo.

```

1  /** Programma che conta le parole in un documento, visualizzando la più frequente. */
2  public class WordCount {
3      public static void main(String[] args) {
4          Map<String, Integer> freq = new ChainHashMap<>(); // o un'altra mappa concreta
5          // legge il documento cercando le parole, usando un delimitatore adeguato
6          Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
7          while (doc.hasNext()) {
8              String word = doc.next().toLowerCase(); // parola convertita in minuscolo
```

```

9   Integer count = freq.get(word); //recupera il conteggio per questa parola
10  if (count == null)
11      count = 0;      // se non è nella mappa, il conteggio precedente è zero
12      freq.put(word, 1 + count); // (ri)assegna il nuovo conteggio a questa parola
13  }
14  int maxCount = 0;
15  String maxWord = "no word";
16  for (Entry<String, Integer> ent : freq.entrySet()) // trova la più frequente
17      if (ent.getValue() > maxCount) {
18          maxWord = ent.getKey();
19          maxCount = ent.getValue();
20      }
21  System.out.print("The most frequent word is '" + maxWord);
22  System.out.println(" with " + maxCount + " occurrences.");
23  }
24 }
```

10.1.3 La classe di base `AbstractMap`

Nel seguito di questo capitolo (e nel prossimo) vedremo molte diverse implementazioni del tipo di dato astratto “mappa”, usando una varietà di strutture dati, ciascuna delle quali mette in atto i propri compromessi tra vantaggi e svantaggi. Come abbiamo fatto nei capitoli precedenti, useremo una combinazione di classi astratte e concrete, con l’obiettivo di un maggior riutilizzo del codice. La Figura 10.2 fornisce un’anticipazione di tali classi.

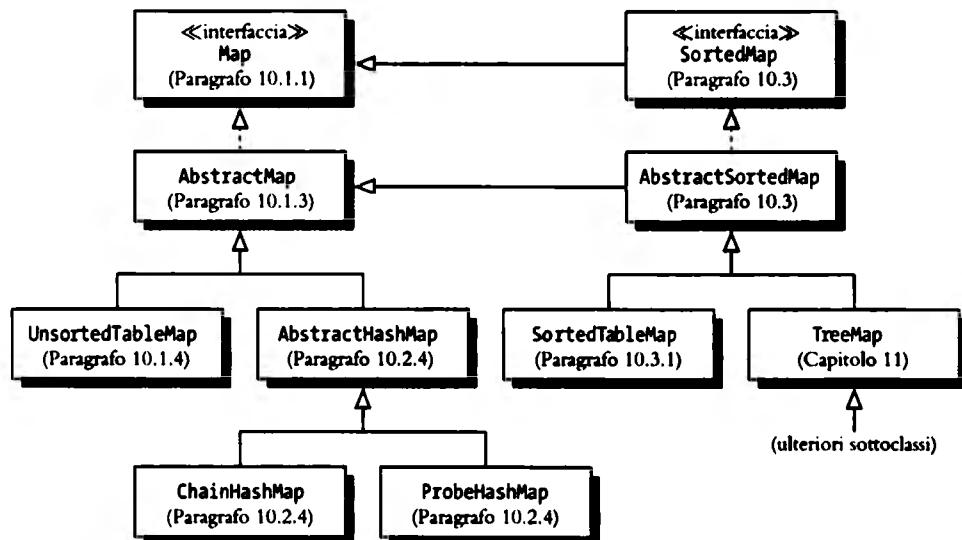


Figura 10.2: La nostra gerarchia di tipi di dati relativi a mappe (con riferimenti alle parti del libro in cui sono definiti).

Iniziamo, in questo paragrafo, progettando una classe di base astratta, `AbstractMap`, che fornisce funzionalità condivise da tutte le nostre implementazioni di mappa. Più specificatamente, la classe di base (descritta nel Codice 10.3) fornisce il seguente supporto:

- Un'implementazione del metodo `isEmpty` basata sulla futura implementazione del metodo `size`.
- Una classe annidata, `MapEntry`, che implementa l'interfaccia pubblica `Entry` e definisce un oggetto composito adatto a memorizzare voci di tipo chiave-valore in una mappa.
- Implementazioni concrete dei metodi `keySet` e `values`, basate su un adattamento del metodo `entrySet`. In questo modo, le classi concrete che implementano una mappa devono soltanto definire il metodo `entrySet` per mettere a disposizione dell'utilizzatore tutte le tre forme di iterazione. Implementiamo le iterazioni usando la tecnica presentata nel Paragrafo 7.4.2 (dove avevamo generato un contenitore iterabile di tutti gli elementi di una lista posizionale dato un contenitore iterabile di tutte le posizioni della lista).

Codice 10.3: Implementazione della classe di base `AbstractMap`.

```

1  public abstract class AbstractMap<K,V> implements Map<K,V> {
2      public boolean isEmpty() { return size() == 0; }
3      //----- classe MapEntry annidata -----
4      protected static class MapEntry<K,V> implements Entry<K,V> {
5          private K k; // chiave
6          private V v; // valore
7          public MapEntry(K key, V value) {
8              k = key;
9              v = value;
10         }
11         // metodi pubblici dell'interfaccia Entry
12         public K getKey() { return k; }
13         public V getValue() { return v; }
14         // metodi ausiliari non facenti parte dell'interfaccia Entry
15         protected void setKey(K key) { k = key; }
16         protected V setValue(V value) {
17             V old = v;
18             v = value;
19             return old;
20         }
21     } //--- fine della classe MapEntry annidata -----
22
23     // supporto per il metodo pubblico keySet...
24     private class KeyIterator implements Iterator<K> {
25         private Iterator<Entry<K,V>> entries = entrySet().iterator();
26         public boolean hasNext() { return entries.hasNext(); }
27         public K next() { return entries.next().getKey(); } // restituisce la chiave!
28         public void remove() { throw new UnsupportedOperationException(); }
29     }
30     private class KeyIterable implements Iterable<K> {
31         public Iterator<K> iterator() { return new KeyIterator(); }
32     }
33     public Iterable<K> keySet() { return new KeyIterable(); }
34
35     // supporto per il metodo pubblico values...
36     private class ValueIterator implements Iterator<V> {
37         private Iterator<Entry<K,V>> entries = entrySet().iterator();
38         public boolean hasNext() { return entries.hasNext(); }
39         public V next() { return entries.next().getValue(); } // restituisce il valore!
```

```

40     public void remove() { throw new UnsupportedOperationException(); }
41 }
42     private class ValueIterable implements Iterable<V> {
43         public Iterator<V> iterator() { return new ValueIterator(); }
44     }
45     public Iterable<V> values() { return new ValueIterable(); }
46 }

```

10.1.4 Una semplice implementazione di mappa non ordinata

Iniziamo a utilizzare la classe `AbstractMap` progettando una prima semplice implementazione concreta dell'ADT mappa che si basa sulla memorizzazione delle coppie chiave-valore in ordine arbitrario all'interno di un esemplare di `ArrayList`. Il Codice 10.4 e 10.5 presenta, appunto, tale classe `UnsortedTableMap`.

Ciascuno dei metodi fondamentali della mappa, `get(k)`, `put(k, v)` e `remove(k)`, inizia con una scansione della lista, per determinare se esista una voce avente chiave uguale a `k`. Per questo motivo abbiamo definito un metodo ausiliario non pubblico, `findIndex(k)`, che restituisce l'indice in cui ha trovato tale voce, oppure `-1` se non l'ha trovata.

Il resto dell'implementazione è piuttosto semplice. Un dettaglio che val la pena menzionare riguarda la rimozione di una voce dalla lista. Anche se potremmo invocare il metodo `remove` della classe `ArrayList`, questo darebbe luogo a un ciclo inutile, allo scopo di far scorrere verso sinistra tutte le voci della lista successive a quella rimossa. Dato che la lista non è ordinata, è preferibile riempire la cella dell'array rimasta vuota spostandovi l'ultima voce: in questo modo, questa fase di modifica avviene in un tempo costante.

Sfortunatamente, la classe `UnsortedTableMap` non è complessivamente molto efficiente. In una mappa avente n voci, ognuno dei metodi fondamentali richiede un tempo d'esecuzione $O(n)$ nel caso peggiore, per la necessità di effettuare una scansione dell'intera lista alla ricerca di una voce presente nella lista. Fortunatamente, nel prossimo paragrafo vedremo una strategia molto più veloce per implementare il tipo di dato astratto "mappa".

Codice 10.4: Un'implementazione di mappa che usa un esemplare di `ArrayList` della libreria standard come tabella non ordinata. L'implementazione prosegue nel Codice 10.5, mentre la superclasse `AbstractMap` è definita nel Codice 10.3.

```

1  public class UnsortedTableMap<K,V> extends AbstractMap<K,V> {
2      /** Array interno che memorizza le voci della mappa. */
3      private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
4
5      /** Costruisce una mappa inizialmente vuota. */
6      public UnsortedTableMap() { }
7
8      // metodo ausiliario privato
9      /** Restituisce l'indice della voce con chiave key, o -1 se non la trova. */
10     private int findIndex(K key) {
11         int n = table.size();
12         for (int j=0; j < n; j++)
13             if (table.get(j).getKey().equals(key))
14                 return j;
15         return -1; // valore speciale per segnalare che la chiave non è stata trovata
16     }

```

Codice 10.5: Un'implementazione di mappa che usa un esemplare di `ArrayList` della libreria standard come tabella non ordinata (continua dal Codice 10.4).

```

17  /** Restituisce il numero di voci presenti nella mappa. */
18  public int size() { return table.size(); }
19  /** Restituisce il valore associato alla chiave specificata (o null se non c'è). */
20  public V get(K key) {
21      int j = findIndex(key);
22      if (j == -1) return null; // non c'è
23      return table.get(j).getValue();
24  }
25  /** Associa il valore alla chiave, sostituendo il valore precedente (se c'era). */
26  public V put(K key, V value) {
27      int j = findIndex(key);
28      if (j == -1) {
29          table.add(new MapEntry<>(key, value)); // aggiunge una nuova voce
30          return null;
31      } else // la chiave è già presente
32          return table.get(j).setValue(value); // restituisce il valore sostituito
33  }
34  /** Elimina la voce con la chiave specificata (se c'è) e restituisce il valore. */
35  public V remove(K key) {
36      int j = findIndex(key);
37      int n = size();
38      if (j == -1) return null; // non c'è
39      V answer = table.get(j).getValue();
40      if (j != n - 1)
41          table.set(j, table.get(n-1)); // sposta l'ultima voce nel "buco" creato
42      table.remove(n-1); // elimina l'ultima voce dalla tabella
43      return answer;
44  }
45  // supporto per il metodo pubblico entrySet...
46  private class EntryIterator implements Iterator<Entry<K,V>> {
47      private int j=0;
48      public boolean hasNext() { return j < table.size(); }
49      public Entry<K,V> next() {
50          if (j == table.size()) throw new NoSuchElementException();
51          return table.get(j++);
52      }
53      public void remove() { throw new UnsupportedOperationException(); }
54  }
55  private class EntryIterable implements Iterable<Entry<K,V>> {
56      public Iterator<Entry<K,V>> iterator() { return new EntryIterator(); }
57  }
58  /** Restituisce un contenitore iterabile con tutte le voci della mappa. */
59  public Iterable<Entry<K,V>> entrySet() { return new EntryIterable(); }
60 }
```

10.2 Tabelle hash

In questo paragrafo presentiamo una delle strutture dati più efficienti per implementare una mappa, che è anche quella effettivamente più utilizzata: la *tabella hash (hash table)*.

Dal punto di vista intuitivo, una mappa M fornisce supporto per un'astrazione che utilizza le chiavi come "indirizzi" che aiutano a localizzare una voce. Come attività di

riscaldamento (mentale), pensiamo a una configurazione limitata, nella quale una mappa con n voci usa chiavi che sono numeri interi compresi in un intervallo che va da 0 a $N - 1$, per qualche valore di $N \geq n$. In questo caso, possiamo rappresentare la mappa usando una cosiddetta *tavella di ricerca (lookup table)* di lunghezza N , come si può vedere nella Figura 10.3.

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Figura 10.3: Una tavella di ricerca (*lookup table*) di lunghezza 11 che rappresenta una mappa contenente le voci (1,D), (3,Z), (6,C) e (7,Q).

In questa rappresentazione, memorizziamo il valore associato alla chiave k nella posizione di indice k della tabella (nell'ipotesi di avere un modo per rappresentare il concetto di cella vuota) e, di conseguenza, le operazioni fondamentali della mappa, *get*, *put* e *remove*, vengono eseguite in un tempo $O(1)$ nel caso peggiore.

Due sono i problemi che nascono quando si vuole estendere questa infrastruttura al caso di mappa più generale. Innanzitutto, non sempre siamo disposti a dedicare all'array uno spazio di N celle, in particolare quando $N \gg n$. Poi, in generale non vogliamo vincolare la mappa a usare numeri interi come chiavi. Il concetto innovativo che sta alla base della tabella hash è l'utilizzo di una *funzione di hash (hash function)* per mettere in corrispondenza chiavi generiche della mappa con indici di una tabella. Idealmente, le chiavi dovrebbero essere ben distribuite dalla funzione di hash nell'intervallo che va da 0 a $N - 1$, ma, in pratica, è possibile che due o più chiavi vengano messe in corrispondenza con uno stesso indice. Di conseguenza, immagineremo la nostra tabella come un “array di contenitori” o *bucket* (“secchio”), visibile nella Figura 10.4, nel quale ogni *bucket* è un contenitore di voci che sono state associate a uno specifico indice dalla funzione di hash (per risparmiare spazio in memoria, un *bucket* vuoto può essere sostituito da un riferimento *null*).

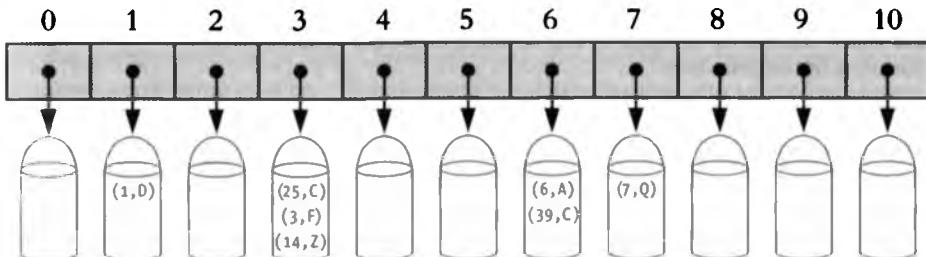


Figura 10.4: Un array di bucket di lunghezza 11 che rappresenta una mappa contenente le voci (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C) e (7,Q), usando una funzione di hash semplice.

10.2.1 Funzioni di hash

L'obiettivo di una *funzione di hash*, h , è quello di mettere in corrispondenza ciascuna chiave k con un numero intero appartenente all'intervallo $[0, N - 1]$, dove N è la capacità dell'array

di bucket di una tabella hash. Individuata una tale funzione, h , l'idea su cui si basa questo approccio è quella di utilizzare il valore calcolato dalla funzione, $h(k)$, come indice nell'array di bucket, A , al posto della chiave k (che può non essere adatta all'uso diretto come indice). Memorizziamo, cioè, la voce (k, v) nel bucket $A[h(k)]$.

Se esistono due o più chiavi che hanno lo stesso *valore di hash*, allora due voci diverse verranno inserite nello stesso bucket in A . In questo caso, diciamo che si è verificata una **collisione**. Esistono diverse strategie per gestire le collisioni, di cui parleremo più avanti, ma la migliore strategia è quella di cercare, per prima cosa, di evitare che si verifichino. A questo proposito, diciamo che una funzione di hash è “buona” se genera un numero di collisioni abbastanza basso. Per motivi pratici, vorremmo anche che la funzione di hash fosse semplice e veloce da calcolare.

È utile considerare la valutazione di una funzione di hash, $h(k)$, come costituita da due fasi: una **codifica di hash**, che mette in corrispondenza una chiave k con un numero intero, e una **funzione di compressione**, che trasforma il codice di hash ottenuto nella prima fase in un numero intero appartenente all'intervallo di indici $[0, N - 1]$, utilizzabile in un array di bucket di dimensione N (si veda la Figura 10.5).

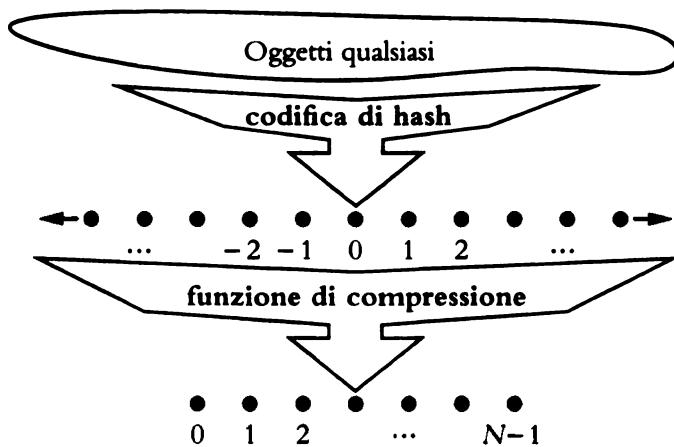


Figura 10.5: Le due componenti di una funzione di hash: la codifica di hash e la funzione di compressione.

Il vantaggio che deriva dalla scomposizione della funzione di hash in tali due componenti sta nel fatto che la parte di calcolo che serve a determinare il codice di hash è indipendente dalla specifica dimensione della tabella hash. Questo consente lo sviluppo, per ciascun tipo di oggetti, di una codifica di hash di applicabilità generale, da poter cioè utilizzare con tabelle hash di qualsiasi dimensione: soltanto la funzione di compressione dipende dalla dimensione della tabella. Si tratta di una strategia particolarmente utile, perché l'array di bucket usato internamente da una tabella hash può essere ridimensionato dinamicamente, in dipendenza dal numero di voci memorizzate nella mappa (si veda il Paragrafo 10.2.3).

Codifiche di hash

La prima azione eseguita da una funzione di hash è quella di agire su una chiave k che appartiene o apparterrà alla mappa e calcolare un numero intero che viene chiamato *codice di hash* di k : non c'è bisogno che questo numero intero appartenga all'intervallo $[0, N - 1]$, può addirittura essere negativo. Ciò che vogliamo è che l'insieme dei codici di hash assegnati alle nostre chiavi eviti al massimo le collisioni, perché se i codici di hash di due chiavi generano una collisione, non c'è alcuna speranza che la funzione di compressione la eviti. Partiamo da una discussione teorica sulla codifica di hash, per proseguire con alcune implementazioni pratiche di codifiche di hash in Java.

Stringa di bit considerata come numero intero

Per iniziare, osserviamo che, per qualsiasi tipo di dato X che sia rappresentato usando al massimo un numero di bit uguale a quello utilizzato per i numeri interi che costituiscono la nostra codifica di hash, possiamo semplicemente usare come codice di hash di X il numero intero che si ottiene interpretando i suoi bit come se rappresentassero, appunto, un numero intero. Java usa codici di hash a 32 bit, quindi per i tipi fondamentali `byte`, `short`, `int` e `char` possiamo calcolare un buon codice di hash facendo semplicemente un cast per ottenere un numero di tipo `int`. Analogamente, nel caso di una variabile x del tipo fondamentale `float`, possiamo convertire x in un numero intero usando l'invocazione `Float.floatToIntBits(x)`, usando poi tale numero intero come codice di hash di x .

Per un tipo di dato la cui rappresentazione come stringa di bit è più lunga di quella del codice di hash voluto (come, ad esempio, avviene per i tipi di dati `long` e `double` in Java), lo schema precedente non è immediatamente applicabile. Una possibilità consiste nell'utilizzo dei soli 32 bit più significativi (o meno significativi) della rappresentazione. Ovviamente questo codice di hash ignora metà dell'informazione presente nella chiave originaria e se molte delle chiavi della mappa differiscono soltanto nei bit ignorati, usando questa semplice codifica di hash colideranno.

Un approccio migliore prevede di combinare in qualche modo le porzioni dei 32 bit più significativi e meno significativi di una chiave a 64 bit formando un codice di hash a 32 bit, che tiene in considerazione tutti i bit originari. Una semplice implementazione di questa strategia consiste nell'addizionare le due componenti come se fossero numeri di 32 bit (ignorando l'eventuale riporto che genera *overflow*) o nel prendere l'or esclusivo (*exclusive-or*) delle due componenti, bit per bit. Questi approcci che combinano componenti si possono estendere a qualunque oggetto x la cui rappresentazione binaria può essere vista come una n -upla di numeri interi a 32 bit, $(x_0, x_1, \dots, x_{n-1})$, generando, ad esempio, il codice di hash come $\Sigma_i x_i$ oppure come $x_0 \oplus x_1 \oplus \dots \oplus x_{n-1}$, dove il simbolo \oplus rappresenta l'operazione di or esclusivo bit per bit (che, in Java, è l'operatore `^`).

Codifiche di hash polinomiali

Le codifiche di hash mediante sommatoria o or esclusivo, appena descritte, non sono scelte buone per stringhe di caratteri o altri oggetti di lunghezza variabile che possano essere considerati come n -uple aventi il formato $(x_0, x_1, \dots, x_{n-1})$, dove l'ordine posizionale tra le componenti x_i è rilevante. Consideriamo, ad esempio, una codifica di hash a 16 bit per una stringa di caratteri s che sommi i valori dei codici Unicode dei caratteri di s . Questo codice di hash produce sfortunatamente molte collisioni indesiderate per gruppi di stringhe molto diffuse. In particolare, ad esempio, usando tale funzione "temp01" e "temp10" collidono,

così come collidono le stringhe "stop", "tops", "pots" e "spot". Una codifica di hash migliore dovrebbe in qualche modo tenere in considerazione le posizioni delle componenti x_i . Un codice di hash alternativo, che fa esattamente questo, prevede di scegliere una costante diversa da zero, $a \neq 1$, e di usare come codice di hash il valore

$$x_0a^{n-1} + x_1a^{n-2} + \cdots + x_{n-2}a + x_{n-1}.$$

In termini matematici, questo è semplicemente un polinomio in a che usa come propri coefficienti le componenti $(x_0, x_1, \dots, x_{n-1})$ della rappresentazione binaria di un oggetto x . Questa codifica di hash viene quindi chiamato **codifica di hash polinomiale**. Per la regola di Horner (si veda l'Esercizio C-4.54), il valore di tale polinomio può essere calcolato come:

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x_1 + ax_0)) \cdots)).$$

Dal punto di vista intuitivo, una codifica di hash polinomiale usa la moltiplicazione per potenze diverse come strumento per distribuire l'influenza di ciascun componente sul codice di hash risultante.

Ovviamente, in un calcolatore la valutazione di una funzione polinomiale verrà effettuata usando una rappresentazione finita per i codici di hash, quindi il valore calcolato potrà non trovar posto nei bit usati per rappresentare un numero intero, generando overflow. Dato che siamo interessati solamente a una buona distribuzione dell'oggetto x rispetto alle altre chiavi, possiamo semplicemente ignorare queste situazioni di overflow, anche se dobbiamo comunque essere consapevoli che tali overflow accadono e, quindi, conviene scegliere una costante a in modo che abbia qualche bit diverso da zero tra quelli meno significativi, per preservare una parte del contenuto informativo della chiave anche se si viene a creare una situazione di overflow.

Abbiamo condotto alcuni studi sperimentali che suggeriscono 33, 37, 39 e 41 come valori particolarmente buoni di a quando si opera con stringhe di caratteri appartenenti al vocabolario della lingua inglese. Infatti, dato un elenco di più di 50 mila parole inglesi formate dall'unione dei dizionari usati in due varianti del sistema operativo Unix, abbiamo scoperto che ciascuno di tali valori di a produce meno di 7 collisioni!

Codifiche di hash per scorrimento ciclico

Una variante della codifica di hash polinomiale prevede di sostituire la moltiplicazione per a con lo scorrimento ciclico di una somma parziale per un certo numero di posizioni. Ad esempio, lo scorrimento ciclico di 5 posizioni del valore a 32 bit 00111101100101101010100010101010100010101000000111. Anche se questa operazione non ha un significato immediatamente riconoscibile in termini aritmetici, raggiunge l'obiettivo di mescolare bene i bit durante il calcolo. In Java, lo scorrimento ciclico di una configurazione di bit si può ottenere facendo un uso attento degli operatori di scorrimento bit per bit.

In Java, una possibile implementazione di una codifica di hash per scorrimento ciclico relativa a una stringa di caratteri potrebbe essere questa:

```
static int hashCode(String s) {
    int h=0;
```

```

for (int i=0; i < s.length(); i++) {
    h = (h << 5) | (h >> 27); // scorrimento ciclico di 5 posizioni
    h += (int) s.charAt(i);    // somma un nuovo carattere
}
return h;
}

```

Come con la codifica di hash polinomiale, anche con la codifica di hash per scorrimento ciclico è necessario determinare con cura il valore del parametro coinvolto, che in questo caso è l'ampiezza dello scorrimento eseguito dopo l'addizione di ciascun carattere. La nostra scelta di uno scorrimento di 5 posizioni è giustificata da esperimenti eseguiti su un elenco di più di 230 mila parole inglesi, confrontando il numero di collisioni ottenute per vari valori dello scorrimento (come si può vedere nella Tabella 10.1).

Tabella 10.1: Confronto tra le collisioni ottenute con la codifica di hash per scorrimento ciclico applicata a un elenco di 230 mila parole inglesi. La colonna "Totale" riporta il numero totale di parole che collidono con almeno un'altra parola, mentre la colonna "Massimo" segnala il numero massimo di parole che collidono in un unico codice di hash. Si noti che usando uno scorrimento uguale a 0 questa codifica di hash degenera in quella che addiziona semplicemente tutti i caratteri.

Scorrimento	Collisioni	
	Totale	Massimo
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

Codifica di hash in Java

Il concetto di codifica di hash è parte integrante del linguaggio Java. La classe `Object`, che funge da antenato per tutti i tipi di oggetti, contiene un metodo, `hashCode()`, che restituisce un numero intero a 32 bit di tipo `int` che può svolgere il ruolo di codice di hash di un oggetto. La versione predefinita di `hashCode()` messa a disposizione dalla classe `Object` è spesso solamente una rappresentazione sotto forma di numero intero dell'indirizzo dell'oggetto in memoria.

Quando si progetta una classe e si pensa di affidarsi alla versione predefinita di `hashCode()`, bisogna fare attenzione. Perché gli schemi che usano una codifica di hash siano affidabili, è necessario che due oggetti che sono considerati “uguali” abbiano lo stesso codice di hash. Questo è importante, perché, se una voce viene inserita in una mappa e, più tardi, la si cerca fornendo come parametro una chiave che l'applicazione considera equivalente alla chiave presente nella voce, la mappa deve essere in grado di individuare tale corrispondenza (si veda, ad esempio, il metodo `UnsortedTableMap.findIndex` nel Codice 10.4). Quindi, quando implementiamo una mappa usando una tabella hash, vogliamo che chiavi equivalenti abbiano lo stesso codice di hash, in modo che la mappa le metta in corrispondenza con lo stesso bucket. Detto più formalmente, se una classe definisce l'equivalenza dei propri esemplari mediante il metodo `equals` (come visto nel Paragrafo 3.5), allora tale classe deve anche definire un'implementazione coerente del metodo `hashCode`, in modo che, se `x.equals(y)`, allora `x.hashCode() == y.hashCode()`.

Ad esempio, la classe `String` della libreria di Java definisce il metodo `equals` in modo che due suoi esemplari siano equivalenti se contengono precisamente la stessa sequenza di caratteri. Tale classe sovrascrive anche il metodo `hashCode` in modo da fornire un comportamento coerente e, in effetti, l'implementazione della codifica di hash nella classe `String` è eccellente. Se ripetiamo l'esperimento citato in precedenza usando l'implementazione della codifica di hash usata da Java, otteniamo soltanto 12 collisioni tra più di 230 mila parole. Anche le classi involucro dei tipi fondamentali di Java definiscono il metodo `hashCode`, usando proprio le tecniche descritte in questo paragrafo.

Come esempio di implementazione appropriata del metodo `hashCode` per una classe che non appartenga alla libreria standard, riprendiamo in esame la classe `SinglyLinkedList`, definita nel Capitolo 3. In quella classe abbiamo definito il metodo `equals`, nel Paragrafo 3.5.2, in modo che due liste siano equivalenti se rappresentano sequenze aventi la stessa lunghezza e contenenti elementi uguali in posizioni corrispondenti. Possiamo calcolare un codice di hash coerente per una tale lista facendo l'or esclusivo dei codici di hash dei suoi elementi, mentre eseguiamo uno scorrimento ciclico (come si può vedere nel Codice 10.6).

Codice 10.6: Un'implementazione coerente del metodo `hashCode` per la classe `SinglyLinkedList` definita nel Capitolo 3.

```

1 public int hashCode() {
2     int h = 0;
3     for (Node walk=head; walk != null; walk = walk.getNext()) {
4         h ^= walk.getElement().hashCode(); // or esclusivo bit per bit con il codice
5         h = (h << 5) | (h >> 27);           // scorrimento ciclico di 5 posti
6     }
7     return h;
8 }
```

Funzioni di compressione

Tipicamente il codice di hash calcolato per una chiave k non sarà già adatto a un utilizzo in un array di bucket, perché il numero intero potrebbe essere negativo o potrebbe eccedere la capacità dell'array di bucket. Di conseguenza, dopo aver determinato un codice di hash intero per un oggetto k di tipo chiave, rimane il problema di mettere in corrispondenza tale numero intero con un (altro) numero intero appartenente all'intervallo $[0, N - 1]$. Questa elaborazione, che prende il nome di *funzione di compressione*, è la seconda fase dell'elabo-

razione complessiva svolta da una funzione di hash. Una buona funzione di compressione rende minimo il numero di collisioni, dato un insieme di codici di hash distinti.

Il metodo della divisione

Il *metodo della divisione* è una semplice funzione di compressione, che trasforma un numero intero i nel numero

$$i \bmod N,$$

dove N , la dimensione dell'array di bucket, è un numero intero positivo prefissato. Inoltre, se scegliamo N in modo che sia un numero primo, questa funzione di compressione "distribuisce" bene i valori che genera. In effetti, se N non è un numero primo, si corre un forte rischio: schemi ripetitivi presenti nella distribuzione dei codici di hash verranno ripetuti nella distribuzione dei valori generati, provocando collisioni. Se, ad esempio, inseriamo chiavi con codici di hash $\{200, 205, 210, 215, 220, \dots, 600\}$ in un array di bucket di dimensione 100, ogni codice di hash entrerà in collisione con parecchi altri codici, ma se usiamo un array di bucket di dimensione 101, non ci sarà alcuna collisione. Se una funzione di hash è stata scelta bene, deve garantire che la probabilità che due chiavi diverse vadano a finire nello stesso bucket sia $1/N$. Tuttavia, scegliere N in modo che sia un numero primo non è sempre sufficiente, perché, se tra i codici di hash esiste uno schema ripetitivo avente la forma $pN + q$ per parecchi valori diversi di p , allora ci saranno comunque collisioni.

Il metodo MAD

Una funzione di compressione più sofisticata, che aiuta a eliminare schemi ripetitivi presenti in un insieme di codici di hash interi, è il metodo *Moltiplica-Somma-e-Dividì* (MAD, *Multiply-Add-and-Divide*), che trasforma il numero intero i in questo valore:

$$[(ai + b) \bmod p] \bmod N,$$

dove N è la dimensione dell'array di bucket, p è un numero primo maggiore di N e a e b sono numeri interi scelti a caso nell'intervallo $[0, p - 1]$, con $a > 0$. Questa funzione di compressione viene solitamente scelta per eliminare schemi ripetitivi presenti nell'insieme dei codici di hash e si avvicina a una funzione di hash "buona", cioè una funzione tale che la collisione tra due chiavi diverse sia $1/N$, il comportamento che si avrebbe se le stesse chiavi venissero "gettate" in A casualmente, con distribuzione uniforme.

10.2.2 Schemi di gestione delle collisioni

L'idea principale su cui si basa una tabella hash è quella di prendere un array di bucket, A , e una funzione di hash, h , per implementare una mappa memorizzando ciascuna voce di tipo (k, v) nel "bucket" identificato da $A[h(k)]$. Questa semplice idea, però, diventa problematica quando esistono due chiavi distinte, k_1 e k_2 , tali che $h(k_1) = h(k_2)$. L'esistenza di tali collisioni ci impedisce di inserire semplicemente una nuova voce (k, v) direttamente nella cella $A[h(k)]$ e complica anche le procedure che eseguono le operazioni di inserimento, ricerca e rimozione.

Concatenazione separata

Un modo efficiente e semplice per gestire le collisioni consiste nel fare in modo che ogni bucket $A[j]$ memorizzi un proprio contenitore secondario, destinato ad accogliere tutte le voci (k, v) tali che $h(k) = j$. Una scelta naturale per tale contenitore secondario è un piccolo esemplare di mappa, implementata usando una lista non ordinata, come visto nel Paragrafo 10.1.4. Questa regola di *risoluzione delle collisioni* prende il nome di *concatenazione separata* (*separate chaining*) ed è illustrata nella Figura 10.6.

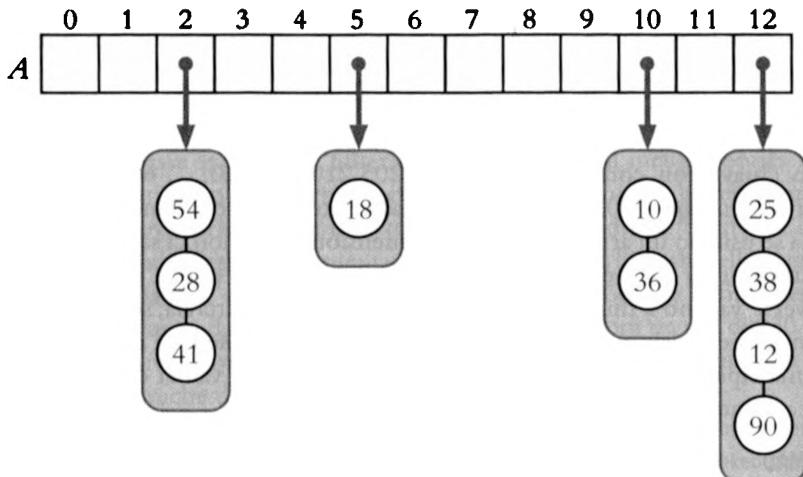


Figura 10.6: Una tabella hash di dimensione 13 che memorizza 10 voci aventi numeri interi come chiavi, con collisioni risolte mediante concatenazione separata. La funzione di compressione è $h(k) = k \bmod 13$. Per semplicità non mostriamo i valori associati alle chiavi.

Nel caso peggiore, le operazioni che riguardano un singolo bucket richiedono un tempo proporzionale alla dimensione del bucket stesso. Nell'ipotesi che si usi una buona funzione di hash per assegnare le n voci di una mappa ai bucket di un array di capacità N , la dimensione attesa di ciascun bucket è n/N . Quindi, data una buona funzione di hash, le operazioni fondamentali della mappa vengono eseguite in un tempo $O(n/N)$. Il rapporto $\lambda = n/N$, chiamato *fattore di occupazione* o di carico (*load factor*) della tabella hash, dovrebbe essere limitato superiormente da un valore piccolo e costante, preferibilmente minore di 1. Se λ è $O(1)$, le operazioni fondamentali della tabella hash vengono eseguite in un tempo atteso che è $O(1)$.

Indirizzamento aperto

La regola che governa la concatenazione separata ha molte proprietà interessanti, come il fatto che consente di implementare in modo semplice le operazioni caratteristiche di una mappa, ma ha senza dubbio anche un piccolo svantaggio: richiede l'uso di strutture dati ausiliarie per memorizzare le voci le cui chiavi collidono. Se lo spazio è importante (ad esempio, se si sta scrivendo un programma destinato a un piccolo dispositivo portatile), allora possiamo adottare un approccio alternativo, che prevede di memorizzare ogni voce direttamente in una cella della tabella. Questo approccio risparmia spazio perché non

utilizza alcuna struttura ausiliaria, ma richiede codice un po' più complesso per gestire in modo adeguato le collisioni. Esistono molte varianti di questa idea e si parla genericamente di schemi a *indirizzamento aperto* (*open addressing*). L'indirizzamento aperto richiede che il fattore di occupazione sia sempre non superiore a 1 e che le voci vengano memorizzate direttamente nelle celle dell'array.

Esplorazione lineare e sue varianti

Un semplice metodo che consente di gestire le collisioni con l'indirizzamento aperto è l'*esplorazione lineare* (*linear probing*). In questo approccio, se cerchiamo di inserire una voce (k, v) in un bucket $A[j]$ che è già occupato, con $j = h(k)$, allora il tentativo successivo riguarderà il bucket $A[(j + 1) \bmod N]$. Se anche quello è già occupato, passiamo a $A[(j + 2) \bmod N]$, e così via, fino a trovare un bucket vuoto in cui poter memorizzare la nuova voce. Dopo aver individuato tale bucket, vi inseriamo semplicemente la voce. Ovviamente questa strategia di risoluzione delle collisioni richiede che venga modificata anche l'implementazione della procedura di ricerca di una chiave nella mappa, che costituisce la prima fase di tutte le operazioni fondamentali: *get*, *put* e *remove*. In particolare, per cercare di localizzare una voce avente chiave uguale a k , dobbiamo esaminare celle consecutive, a partire da $A[h(k)]$, finché non troviamo una voce avente chiave uguale a k oppure scopriamo un bucket vuoto (si veda la Figura 10.7). Il nome "esplorazione lineare" deriva proprio dal fatto che l'accesso a una cella dell'array di bucket può essere visto come una "esplorazione" e che esplorazioni consecutive avvengono in celle adiacenti, secondo uno schema lineare (quando l'array viene considerato circolare).



Figura 10.7: Inserimento in una tabella hash che usa l'esplorazione lineare con chiavi intere. La funzione di hash è $h(k) = k \bmod 11$. I valori associati alle chiavi non sono mostrati nella figura.

Per effettuare una rimozione, non possiamo semplicemente eliminare la voce che abbiamo trovato rendendo vuota la cella che occupa nell'array. Ad esempio, dopo l'inserimento della chiave 15 descritto nella Figura 10.7, se cancellassimo banalmente la voce che ha chiave 37, una successiva ricerca della chiave 15 fallirebbe, perché inizierebbe esplorando la cella di indice 4, per passare alla cella di indice 5 e, infine, alla cella di indice 6, che troverebbe vuota. Una strategia tipicamente utilizzata per risolvere questo problema consiste nel sostituire la voce rimossa con uno speciale oggetto sentinella. Quando questa sentinella occupa spazi nella nostra tabella hash, dobbiamo modificare l'algoritmo di ricerca in modo che, cercando una chiave k , salti le celle che contengono la sentinella, continuando l'esplorazione fino a raggiungere la voce cercata o un bucket vuoto (accorgendosi dell'eventuale ritorno alla posizione esplorata inizialmente, un'altra condizione che segnala l'assenza della chiave cer-

cata). Inoltre, se stiamo eseguendo un'operazione `put`, la ricerca di k deve ricordarsi di aver eventualmente visto una sentinella, perché quella posizione è valida per inserire la nuova voce (k, v) , se non viene trovata una voce avente chiave k .

Anche se l'uso di uno schema a indirizzamento aperto può far risparmiare spazio, l'esplorazione lineare porta con sé un nuovo svantaggio: tende a raggruppare le voci della mappa in serie di celle contigue (un fenomeno detto *clustering*), che possono anche sovrapporsi (in modo particolare se più di metà delle celle della tabella hash sono state occupate). Tali serie di celle contigue occupate rallentano notevolmente le ricerche.

Una diversa strategia di indirizzamento aperto, l'*esplorazione quadratica* (*quadratic probing*), controlla in sequenza i bucket $A[(h(k) + f(i)) \bmod N]$, con $i = 0, 1, 2, \dots$, e $f(i) = i^2$, finché non trova un bucket vuoto. Come con l'esplorazione lineare, anche con l'esplorazione quadratica l'operazione di rimozione si complica, ma si evita la formazione di sequenze di celle occupate, che invece affliggono l'esplorazione lineare. Anche quest'ultima strategia, però, genera un proprio tipo di *clustering*, chiamato *clustering secondario*, nel quale l'insieme di celle piene ha uno schema non omogeneo, anche quando i codici di hash originari sono distribuiti uniformemente. Quando N è un numero primo e l'array di bucket è pieno per meno della metà, l'esplorazione quadratica garantisce di trovare una cella vuota. Questa garanzia, però, non vale quando la tabella è piena almeno per metà, oppure se N non è un numero primo: analizzeremo la causa di questo tipo di clustering nell'Esercizio C-10.42.

Un approccio di indirizzamento aperto che non provoca fenomeni di clustering come quelli visti per l'esplorazione lineare o l'esplorazione quadratica è il *doppio hashing* (*double hashing*). In questa strategia scegliamo una funzione di hash secondaria, h' , e, quando h assegnerebbe una chiave k al bucket $A[h(k)]$ già occupato, esploriamo in sequenza i bucket $A[(h(k) + f(i)) \bmod N]$, con $i = 1, 2, 3, \dots$, e $f(i) = i \cdot h'(k)$. In questo schema, la funzione di hash secondaria non può assumere il valore zero; una scelta frequente è $h'(k) = q - (k \bmod q)$, essendo N un numero primo e q un numero primo con $q < N$.

Un ultimo approccio che proponiamo per evitare il fenomeno del clustering con l'indirizzamento aperto prevede di esplorare in sequenza i bucket $A[(h(k) + f(i)) \bmod N]$, con $f(i)$ funzione basata su un generatore di numeri pseudocasuali che fornisca una sequenza ripetibile, ma in qualche modo arbitraria, di celle da esplorare che dipenda dai bit del codice di hash originario.

10.2.3 Fattore di carico, rehashing ed efficienza

Negli schemi di tabella hash descritti finora, è importante che il fattore di carico, $\lambda = n/N$, sia tenuto al di sotto di 1. Con la concatenazione separata, quando λ si avvicina molto a 1, la probabilità di collisione aumenta molto, aggiungendo tempo d'esecuzione alle operazioni, perché nei bucket in cui avvengono collisioni bisogna passare a metodi basati su una lista, che richiedono un tempo d'esecuzione lineare. Gli esperimenti e le analisi del caso medio suggeriscono che nelle tabelle hash che usano la concatenazione separata si debba mantenere $\lambda < 0.9$ (l'implementazione di Java, in mancanza di indicazioni diverse, usa la concatenazione separata con $\lambda < 0.75$).

Con l'indirizzamento aperto, d'altra parte, quando il fattore di carico λ supera 0.5 e inizia ad avvicinarsi a 1, le sequenze (*cluster*) di voci nell'array di bucket iniziano analogamente a crescere di dimensione. Questi cluster fanno sì che le strategie di esplorazione "saltino" in qua e in là nell'array di bucket per una quantità di tempo considerevole prima di trovare una

cella vuota. Nell'Esercizio C-10.42 analizziamo come si degradi l'esplorazione quadratica quando $\lambda \geq 0.5$. Gli esperimenti suggeriscono che in uno schema a indirizzamento aperto con esplorazione lineare si dovrebbe mantenere $\lambda < 0.5$, forse un valore solo un po' più alto se si usano altri schemi di indirizzamento aperto.

Se un'operazione di inserimento in una tabella hash fa salire il fattore di carico al di sopra della soglia specificata, solitamente si ridimensiona la tabella stessa (per ripristinare il fattore di carico specificato) e si reinseriscono tutti gli oggetti in questa nuova tabella. Sebbene non sia necessario definire un nuovo codice di hash per ciascun oggetto, bisogna invece sicuramente applicare una nuova funzione di compressione che prenda in considerazione la dimensione della nuova tabella. Questo meccanismo, detto di *rehashing*, in generale distribuirà le voci sull'intero nuovo array di bucket. Quando si applica il rehashing per generare una nuova tabella, è bene che la dimensione del nuovo array sia un numero primo approssimativamente doppio della dimensione precedente (si veda l'Esercizio C-10.32). In tal modo, il costo del reinserimento di tutte le voci nella nuova tabella può essere ammortizzato con il tempo utilizzato per inserirle la prima volta (come per gli array dinamici, secondo quanto visto nel Paragrafo 7.2.1).

Efficienza delle tabelle hash

Anche se i dettagli dell'analisi della strategia di hashing nel caso medio vanno al di là degli obiettivi di questo libro, gli elementi probabilistici su cui si basa sono abbastanza intuitivi. Se la nostra funzione di hash è buona, ci aspettiamo che le voci vengano distribuite uniformemente tra le N celle dell'array di bucket, quindi, per memorizzare n voci, il numero atteso di chiavi all'interno di un bucket è $\lceil n/N \rceil$, una quantità che è $O(1)$ se n è $O(N)$.

I costi associati a un'azione di rehashing periodica (conseguente ai ridimensionamenti della tabella a causa di un certo numero di inserimenti o rimozioni) possono essere tenuti in conto separatamente, aggiungendo alle operazioni *put* e *remove* un ulteriore costo ammortizzato $O(1)$.

Nel caso peggiore, una funzione di hash non buona potrebbe assegnare tutte le voci al medesimo bucket. Questo produrrebbe prestazioni temporali lineari per le operazioni fondamentali della mappa nel caso di concatenazione separata, così come nel caso di un modello a indirizzamento aperto nel quale la sequenza di esplorazioni secondarie dipenda soltanto dal codice di hash. Questi costi sono riassunti nella Tabella 10.2.

Tabella 10.2: Confronto tra i tempi d'esecuzione dei metodi di una mappa realizzata mediante una lista non ordinata (come visto nel Paragrafo 10.1.4) oppure una tabella hash. Indichiamo con n il numero di voci presenti nella mappa e ipotizziamo che l'array di bucket che fornisce il supporto di memorizzazione per la tabella hash sia gestito in modo che la sua capacità sia proporzionale al numero di voci presenti nella mappa.

Metodo	Lista non ordinata	Tabella hash	
		Atteso	Caso peggiore
get	$O(n)$	$O(1)$	$O(n)$
put	$O(n)$	$O(1)$	$O(n)$
remove	$O(n)$	$O(1)$	$O(n)$
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
entrySet, keySet, values	$O(n)$	$O(n)$	$O(n)$

Un aneddoto relativo a hashing e sicurezza informatica

In un lavoro scientifico pubblicato nel 2003, alcuni ricercatori discussero della possibilità di sfruttare le prestazioni del caso peggiore di una tabella hash per provocare un attacco di tipo DoS (*denial of service*, cioè “inibizione del servizio”, un tipo di attacco che sovraccarica un servizio impedendone il normale funzionamento) avente come obiettivo le tecnologie di Internet. Dal momento che molti algoritmi calcolano codici di hash con una funzione deterministica, un attaccante potrebbe pre-calcolare un gran numero di stringhe di lunghezza modesta, in modo che abbiano tutte lo stesso codice di hash a 32 bit (ricordando che, con qualunque schema di hashing che abbiamo descritto, con l’esclusione del *double hashing*, se due chiavi hanno lo stesso codice di hash, saranno indistinguibili al momento della risoluzione della collisione). Questo problema fu portato all’attenzione del gruppo di sviluppo del linguaggio Java, così come dei responsabili dello sviluppo di molti altri linguaggi di programmazione, ma in quel momento fu ritenuto un rischio trascurabile da quasi tutti (complimenti, invece, agli sviluppatori del linguaggio Perl, che fin dal 2003 eliminarono il problema).

Verso la fine del 2011, un altro gruppo di ricercatori presentò un’implementazione di un tale attacco. I server web consentono di inserire in un URL una sequenza di parametri nel formato chiave-valore, usando una sintassi come ?key1=val1&key2=val2&key3=val3. Queste coppie chiave-valore sono stringhe e tipicamente un server web le memorizza in una mappa realizzata mediante tabella hash. Per evitare sovraccarichi, i server prevedono già una limitazione della lunghezza e del numero di tali parametri, ma fanno l’ipotesi che il tempo totale richiesto per il loro inserimento nella mappa sarà lineare nel numero di voci, visto che il tempo atteso per le operazioni è costante. Tuttavia, se tutte le chiavi collidono, gli inserimenti nella mappa richiederanno un tempo quadratico, costringendo il server a una quantità di lavoro eccessiva.

Nel 2012, il gruppo di lavoro denominato OpenJDK annunciò la seguente soluzione: venne distribuita una correzione all’ambiente di sviluppo di Java, concernente la sicurezza (*security patch*), che comprendeva una funzione di hash alternativa che introduce casualità nel calcolo dei codici di hash, rendendo ingestibile la progettazione “a ritroso” (*reverse engineering*) di un insieme di stringhe collidenti. Tuttavia, per evitare di rendere non più funzionante il codice Java esistente, questa nuova caratteristica venne disabilitata nelle impostazioni pre-definite della versione SE 7 dell’ambiente di sviluppo e, quando fosse stata abilitata, sarebbe stata utilizzata soltanto per la generazione di codici hash di stringhe e soltanto quando la dimensione della tabella avesse superato una determinata soglia. Tale hashing avanzato venne infine abilitato nella versione SE 8 per tutti i tipi di dati e in tutti i casi.

10.2.4 Implementazione di tabella hash in Java

In questo paragrafo presentiamo due implementazioni di tabella hash, una che usa la concatenazione separata e un’altra che utilizza l’indirizzamento aperto con esplorazione lineare. Nonostante questi approcci per la risoluzione delle collisioni siano molto diversi, i due algoritmi di hashing presentano molte somiglianze quando vengono considerati ad alto livello. Per tale motivo, estendiamo per prima cosa la classe *AbstractMap* (definita nel Codice 10.3) definendo una nuova classe astratta, *AbstractHashMap* (nel Codice 10.7), che provvede a fornire molte delle funzionalità comuni alle nostre due successive implementazioni di tabella hash.

Partiamo con la presentazione di ciò che questa classe astratta *non fa*: non fornisce alcuna rappresentazione concreta di una tabella di "bucket". Con la concatenazione separata, ciascun bucket sarà una mappa secondaria, mentre con l'indirizzamento aperto non esiste alcun contenitore che svolga concretamente il ruolo di bucket: i bucket sono intrecciati, per effetto delle sequenze di esplorazione. Nel nostro progetto, la classe `AbstractHashMap` presuppone che in ciascuna sottoclasse concreta verranno implementati i seguenti metodi astratti:

- `createTable()`: Questo metodo deve creare una tabella inizialmente vuota avente dimensione uguale all'apposita variabile di esemplare `capacity`.
- `bucketGet(h, k)`: Questo metodo deve avere lo stesso comportamento del metodo pubblico `get`, sapendo però che la chiave `k` viene assegnata al bucket `h` dalla funzione di hash.
- `bucketPut(h, k, v)`: Questo metodo deve avere lo stesso comportamento del metodo pubblico `put`, sapendo però che la chiave `k` viene assegnata al bucket `h` dalla funzione di hash.
- `bucketRemove(h, k)`: Questo metodo deve avere lo stesso comportamento del metodo pubblico `remove`, sapendo però che la chiave `k` viene assegnata al bucket `h` dalla funzione di hash.
- `entrySet()`: Questo metodo scandisce *tutte* le voci della mappa. Non lo fa necessariamente "per bucket", perché nell'indirizzamento aperto i bucket non sono necessariamente disgiunti.

Quello che viene fornito dalla classe `AbstractHashMap` è un supporto matematico, sotto forma di una funzione di compressione che usa una formula MAD con componenti casuali (*randomized MAD*), oltre al supporto per il ridimensionamento automatico della tabella hash quando il fattore di carico supera una determinata soglia.

Il metodo `hashValue` si basa sul codice di hash originario della chiave, così come viene restituito dal suo metodo `hashCode()`, a cui applica una compressione MAD basata su un numero primo e sui parametri `scale` e `shift`, scelti in modo casuale nel costruttore.

Per gestire il fattore di carico, la classe `AbstractHashMap` dichiara una variabile `protected`, `n`, che deve essere uguale al numero di voci presenti nella mappa, ma deve delegare alle sottoclassi il compito di aggiornare tale campo, nei metodi `bucketPut` e `bucketRemove`. Se il fattore di carico della tabella supera 0.5, creiamo una tabella più grande (usando il metodo `createTable`) e reinseriamo tutte le voci in tale nuova tabella (per semplicità, questa implementazione usa tabelle la cui dimensione è $2^k + 1$, con `k` intero, anche se in generale non si tratta di numeri primi).

Codice 10.7: Una classe di base per le nostre successive implementazioni di tabella hash; estende la classe `AbstractMap` definita nel Codice 10.3.

```

1 public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
2     protected int n = 0;           // numero di voci nella mappa
3     protected int capacity;      // lunghezza della tabella
4     private int prime;          // un numero primo

```

```

5   private long scale, shift; // i fattori di scala e di scorrimento per MAD
6   public AbstractHashMap(int cap, int p) {
7       prime = p;
8       capacity = cap;
9       Random rand = new Random();
10      scale = rand.nextInt(prime-1) + 1;
11      shift = rand.nextInt(prime);
12      createTable();
13  }
14  public AbstractHashMap(int cap) { this(cap, 109345121); } // un primo predefinito
15  public AbstractHashMap() { this(17); } // capacità predefinita
16  // metodi pubblici
17  public int size() { return n; }
18  public V get(K key) { return bucketGet(hashValue(key), key); }
19  public V remove(K key) { return bucketRemove(hashValue(key), key); }
20  public V put(K key, V value) {
21      V answer = bucketPut(hashValue(key), key, value);
22      if (n > capacity / 2)          // mantiene il fattore di carico <= 0.5
23          resize(2 * capacity - 1); // (oppure un numero primo vicino)
24      return answer;
25  }
26  // metodi ausiliari privati
27  private int hashValue(K key) {
28      return (int) ((Math.abs(key.hashCode()) * scale + shift) % prime) % capacity;
29  }
30  private void resize(int newCap) {
31      ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
32      for (Entry<K,V> e : entrySet())
33          buffer.add(e);
34      capacity = newCap;
35      createTable(); // sulla base della capacità modificata
36      n = 0;          // il valore verrà ricalcolato reinserendo le voci
37      for (Entry<K,V> e : buffer)
38          put(e.getKey(), e.getValue());
39  }
40  // metodi astratti protected che dovranno essere implementati nelle sottoclassi
41  protected abstract void createTable();
42  protected abstract V bucketGet(int h, K k);
43  protected abstract V bucketPut(int h, K k, V v);
44  protected abstract V bucketRemove(int h, K k);
45 }

```

Concatenazione separata

Nella concatenazione separata, per rappresentare ciascun bucket usiamo un esemplare della classe più semplice, `UnsortedTableMap`, vista nel Paragrafo 10.1.4. Questa tecnica, nella quale si usa una soluzione semplice di un problema per generare una nuova e più articolata soluzione, prende il nome di *bootstrapping* ("innesto"): il vantaggio di usare una mappa anche per i singoli bucket è la facilità con cui si possono delegare le responsabilità dalle operazioni agenti sulla mappa principale alle omonime operazioni applicate al bucket appropriato.

L'intera tabella è, quindi, rappresentata da un array A di dimensione prefissata, i cui elementi sono le mappe secondarie. Ogni cella, $A[h]$, contiene inizialmente un riferimento nullo: creiamo una mappa secondaria soltanto quando per la prima volta una voce viene assegnata, mediante hashing, a quel particolare bucket.

Come regola generale, implementiamo `bucketGet(h, k)` invocando `A[h].get(k)`, `bucketPut(h, k, v)` invocando `A[h].put(k, v)` e `bucketRemove(h, k)` invocando `A[h].remove(k)`. Bisogna, però, fare attenzione, per due motivi.

Innanzitutto, dal momento che abbiamo scelto di lasciare il riferimento `null` nelle celle della tabella finché non è effettivamente necessaria la corrispondente mappa secondaria, ciascuna di queste operazioni fondamentali deve cominciare verificando se `A[h]` contiene `null`. Nel caso di `bucketGet` e `bucketRemove`, se il bucket non esiste ancora, possiamo semplicemente restituire `null`, perché non esiste alcuna voce in corrispondenza della chiave `k`. Nel caso di `bucketPut`, invece, bisogna inserire una nuova voce, per cui, prima di continuare, creiamo un nuovo esemplare di `UnsortedTableMap` e lo memorizziamo in `A[h]`.

Il secondo problema deriva dal fatto che, nell'infrastruttura di cui fa parte `AbstractHashMap`, ogni sottoclasse ha la responsabilità di gestire in modo appropriato la variabile di esemplare `n`, nel momento in cui viene inserita o eliminata una voce. Ricordiamo che, quando si invoca il metodo `put(k, v)` su una mappa, la dimensione di tale mappa aumenta soltanto se la chiave `k` non era già presente nella mappa (altrimenti viene semplicemente modificato il valore memorizzato nella voce corrispondente). Analogamente, un'invocazione del metodo `remove(k)` fa diminuire la dimensione della mappa soltanto quando questa contiene effettivamente una voce con chiave uguale a `k`. Nella nostra implementazione, dobbiamo dedurre che ci sia stato un cambiamento nella dimensione della mappa principale controllando se, prima e dopo una delle operazioni citate, la dimensione della mappa secondaria corrispondente è cambiata.

Il Codice 10.8 definisce completamente la nostra classe `ChainHashMap`, che implementa una tabella hash con concatenazione separata. Nell'ipotesi che la funzione di hash sia buona, una mappa con n voci realizzata con una tabella di capacità N ha una dimensione attesa dei bucket pari a n/N (quello che chiamiamo *fattore di carico*). Quindi, anche se i singoli bucket, implementati con esemplari di `UnsortedTableMap`, non sono particolarmente efficienti, ciascun bucket ha una dimensione attesa pari a $O(1)$, a patto che n sia $O(N)$, come effettivamente è nella nostra implementazione. In questa mappa, perciò, il tempo d'esecuzione atteso per le operazioni `get`, `put` e `remove`, è $O(1)$. Il metodo `entrySet` e, quindi, i metodi correlati `keySet` e `values`, vengono eseguiti in un tempo $O(n + N)$, perché effettuano un ciclo che percorre tutta la tabella (che ha lunghezza N) e tutti i bucket (che hanno una dimensione totale uguale a n).

Codice 10.8: Un'implementazione concreta di mappa mediante tabella hash che usa la concatenazione separata.

```

1  public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
2      // un array di esemplari di UnsortedTableMap che fungono da bucket
3      private UnsortedTableMap<K,V>[] table; // verrà inizializzato da createTable
4      public ChainHashMap() { super(); }
5      public ChainHashMap(int cap) { super(cap); }
6      public ChainHashMap(int cap, int p) { super(cap, p); }
7      /** Crea una tabella vuota con lunghezza uguale alla capacità attuale. */
8      protected void createTable() {
9          table = (UnsortedTableMap<K,V>[]) new UnsortedTableMap[capacity];
10     }
11     /** Restituisce il valore associato alla chiave k nel bucket h, altrimenti null. */
12     protected V bucketGet(int h, K k) {
13         UnsortedTableMap<K,V> bucket = table[h];

```

```

        if (bucket == null) return null;
        return bucket.get(k);
    }
    /** Associa la chiave k al valore v nel bucket h; restituisce il vecchio valore. */
    protected V bucketPut(int h, K k, V v) {
        UnsortedTableMap<K,V> bucket = table[h];
        if (bucket == null)
            bucket = table[h] = new UnsortedTableMap<>();
        int oldSize = bucket.size();
        V answer = bucket.put(k, v);
        n += (bucket.size() - oldSize); // la dimensione può essere aumentata
        return answer;
    }
    /** Elimina la voce avente chiave k dal bucket h (se vi è presente). */
    protected V bucketRemove(int h, K k) {
        UnsortedTableMap<K,V> bucket = table[h];
        if (bucket == null) return null;
        int oldSize = bucket.size();
        V answer = bucket.remove(k);
        n -= (oldSize - bucket.size()); // la dimensione può essere diminuita
        return answer;
    }
    /** Restituisce un contenitore iterabile con tutte le voci della mappa. */
    public Iterable<Entry<K,V>> entrySet() {
        ArrayList<Entry<K,V>> buffer = new ArrayList<>();
        for (int h=0; h < capacity; h++)
            if (table[h] != null)
                for (Entry<K,V> entry : table[h].entrySet())
                    buffer.add(entry);
        return buffer;
    }
}

```

Esplorazione lineare

La nostra implementazione di tabella hash con indirizzamento aperto ed esplorazione lineare, `ProbeHashMap`, è presentata nel Codice 10.9 e 10.10. Per consentire le rimozioni di voci dalla mappa, usiamo la tecnica descritta nel Paragrafo 10.2.2, inserendo una speciale sentinella nelle posizioni della tabella in corrispondenza delle quali è avvenuta una rimozione, in modo da poter distinguere tra questa situazione e una cella che, invece, è sempre stata vuota. A questo scopo, creiamo un esemplare di voce, `DEFUNCT`, che agirà da sentinella (indipendentemente dalla chiave e dal valore che contiene), contrassegnando con il riferimento a tale esemplare le celle da cui è stata eliminata una voce.

L'aspetto più complesso dell'indirizzamento aperto è l'esecuzione della sequenza di esplorazioni nel momento in cui avviene una collisione durante la ricerca di una voce nella mappa, o l'inserimento di una nuova voce. A questo scopo, le tre operazioni principali della mappa si basano su un metodo ausiliario, `findSlot`, che cerca una voce con chiave k nel "bucket" h (cioè usando l'indice h restituito dalla funzione di hash applicata alla chiave k). Quando cerchiamo di recuperare il valore associato a una chiave, dobbiamo continuare l'esplorazione finché non troviamo la chiave, oppure raggiungiamo una cella della tabella che contiene il riferimento `null`: non possiamo fermarci nel momento in cui troviamo il riferimento alla sentinella, `DEFUNCT`, perché questo segnala una posizione che, nel momento in cui la voce che cerchiamo è stata inserita, poteva essere occupata.

Quando una coppia chiave-valore viene inserita nella mappa, dobbiamo per prima cosa cercare se esiste una voce che abbia la stessa chiave, in modo da sovrascriverne il valore: durante l'inserimento, quindi, non dobbiamo fermarci in corrispondenza della sentinella. Se, però, non troviamo nessuna voce con la chiave data, è preferibile riutilizzare la prima posizione che contiene la sentinella, se ne esiste una, posizionandovi la nuova voce da inserire nella tabella. Il metodo `findSlot` agisce proprio in questo modo, proseguendo una ricerca fino a trovare una cella veramente vuota e restituendo l'indice della prima cella in cui sarebbe possibile effettuare un inserimento.

Quando, poi, eliminiamo dalla mappa una voce, all'interno del metodo `bucketRemove`, assegniamo alla cella in cui è avvenuta la rimozione il riferimento alla sentinella, `DEFUNCT`, in base a quanto stabilito dalla nostra strategia.

Codice 10.9: La classe `ProbeHashMap`: un'implementazione concreta di mappa mediante tabella hash che usa l'esplorazione lineare per la risoluzione delle collisioni (prosegue nel Codice 10.10).

```

1  public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
2      private MapEntry<K,V>[] table; // un array di voci (inizialmente tutte null)
3      private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null); // sentinella
4      public ProbeHashMap() { super(); }
5      public ProbeHashMap(int cap) { super(cap); }
6      public ProbeHashMap(int cap, int p) { super(cap, p); }
7      /** Crea una tabella vuota con lunghezza uguale alla capacità attuale. */
8      protected void createTable() {
9          table = (MapEntry<K,V>[][]) new MapEntry[capacity];
10     }
11     /** Restituisce true se e solo se la posizione è vuota o contiene la sentinella. */
12     private boolean isAvailable(int j) {
13         return (table[j] == null || table[j] == DEFUNCT);
14     }

```

Codice 10.10: La classe `ProbeHashMap`: un'implementazione concreta di mappa mediante tabella hash che usa l'esplorazione lineare per la risoluzione delle collisioni (continua dal Codice 10.9).

```

15     /** Restituisce l'indice della chiave o -(a+1) se in a si può inserire k. */
16     private int findSlot(int h, K k) {
17         int avail = -1;           // nessuna posizione disponibile
18         int j = h;               // indice durante la scansione della tabella
19         do {
20             if (isAvailable(j)) { // può essere vuota o con sentinella
21                 if (avail == -1) avail = j; // è la prima posizione disponibile!
22                 if (table[j] == null) break; // è vuota: la ricerca fallisce immediatamente
23             } else if (table[j].getKey().equals(k))
24                 return j;           // la ricerca ha avuto successo
25             j = (j+1) % capacity;   // continua a cercare (ciclicamente)
26         } while (j != h);        // si ferma se torna all'inizio
27         return -(avail + 1);    // la ricerca ha fallito
28     }
29     /** Restituisce il valore associato alla chiave k nel bucket h, altrimenti null. */
30     protected V bucketGet(int h, K k) {
31         int j = findSlot(h, k);
32         if (j < 0) return null;    // chiave non trovata
33         return table[j].getValue();
34     }

```

```

    /** Associa la chiave k al valore v nel bucket h; restituisce il vecchio valore. */
protected V bucketPut(int h, K k, V v) {
    int j = findSlot(h, k);
    if (j >= 0)                                // chiave trovata
        return table[j].setValue(v);
    table[-(j+1)] = new MapEntry<>(k, v); // converte j nell'indice corretto
    n++;
    return null;
}
/** Elimina la voce avente chiave k dal bucket h (se vi è presente). */
protected V bucketRemove(int h, K k) {
    int j = findSlot(h, k);
    if (j < 0) return null;           // chiave non trovata, niente da eliminare
    V answer = table[j].getValue();
    table[j] = DEFUNCT;            // contrassegna la cella come disattivata
    n--;
    return answer;
}
/** Restituisce un contenitore iterabile con tutte le voci della mappa. */
public Iterable<Entry<K,V>> entrySet() {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>();
    for (int h=0; h < capacity; h++)
        if (!isAvailable(h)) buffer.add(table[h]);
    return buffer;
}
}

```

10.3 Mappe ordinate

Tradicionalmente la mappa, come tipo di dato astratto, consente al suo utilizzatore di cercare il valore associato a una determinata chiave, ma la ricerca della chiave viene eseguita sotto forma di *ricerca esatta* (*exact search*). In questo paragrafo vedremo un'estensione di tale ADT che prende il nome di *mappa ordinata* (*sorted map*) e ha tutti i comportamenti della mappa normale, più i seguenti:

- firstEntry():** Restituisce la voce avente la chiave minima (o `null`, se la mappa è vuota).
- lastEntry():** Restituisce la voce avente la chiave massima (o `null`, se la mappa è vuota).
- ceilingEntry(*k*):** Restituisce la voce avente la chiave minima tra quelle non minori di *k* (o `null`, se non esiste).
- floorEntry(*k*):** Restituisce la voce avente la chiave massima tra quelle non maggiori di *k* (o `null`, se non esiste).
- lowerEntry(*k*):** Restituisce la voce avente la chiave massima tra quelle minori di *k* (o `null`, se non esiste).
- higherEntry(*k*):** Restituisce la voce avente la chiave minima tra quelle maggiori di *k* (o `null`, se non esiste).
- subMap(*k₁*, *k₂*):** Restituisce una (sotto)mappa contenente tutte le voci aventi chiave non minore di *k₁* e minore di *k₂*.

Osserviamo che i metodi qui citati fanno tutti parte dell'interfaccia `java.util.NavigableMap` (che estende la più semplice interfaccia `java.util.SortedMap`).

Per motivare l'uso di una mappa ordinata, consideriamo un sistema di elaborazione che gestisca le informazioni relative a eventi avvenuti nel passato (come, ad esempio, transazioni finanziarie), con un *time stamp* (*istante di tempo*) che identifica il momento in cui ciascun evento si è verificato. Se tali "istanti di tempo" fossero unici all'interno di un determinato sistema, potremmo utilizzare una mappa usando gli istanti di tempo come chiavi, associate a un oggetto (*record*) contenente tutte le informazioni relative all'evento, usato come valore. Un particolare istante di tempo potrebbe, quindi, svolgere il ruolo di identificatore (ID) di riferimento per un evento, nel qual caso potremmo recuperare velocemente le informazioni relative a un evento presente nella mappa. Tuttavia, il tipo di dato astratto "mappa (non ordinata)" non fornisce strumenti per ottenere un elenco di tutti gli eventi ordinati in base all'istante di tempo in cui sono avvenuti, né di cercare gli eventi avvenuti in prossimità di un determinato istante di tempo. Infatti, le implementazioni dell'ADT "mappa" basate su tabella hash sparpagliano volutamente le chiavi che possono sembrare molto "simili" nel loro dominio originario, in modo che siano assegnate alle posizioni della tabella hash in modo più uniforme.

10.3.1 Tabelle di ricerca ordinate

Diverse strutture dati sono in grado di offrire un'efficiente implementazione del tipo di dato astratto "mappa ordinata" e alcune tecniche avanzate saranno esaminate in dettaglio nel Paragrafo 10.4 e nel Capitolo 11. In questo paragrafo, invece, analizzeremo un'implementazione di mappa ordinata piuttosto semplice: memorizziamo le voci della mappa in un array A in modo che si trovino in ordine di chiave crescente, come nella Figura 10.8. Chiamiamo questa implementazione *tabella di ricerca ordinata* (*sorted search table*).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Figura 10.8: Realizzazione di una mappa mediante una tabella di ricerca ordinata. Abbiamo riportato soltanto le chiavi, in modo da evidenziare il loro ordinamento.

Come nel caso della mappa memorizzata in una tabella o lista non ordinata, visto nel Paragrafo 10.1.4, la tabella di ricerca ordinata richiede uno spazio di memorizzazione $O(n)$. Il vantaggio principale di questa rappresentazione, e il motivo per cui abbiamo richiesto che A fosse una lista realizzata mediante array, sta nel fatto che consente l'utilizzo dell'algoritmo di ricerca binaria o per bisezione (*binary search*), sfruttandone l'efficienza per molte diverse operazioni.

Ricerca binaria e ricerca di corrispondenze non esatte

La nostra prima trattazione dell'algoritmo di ricerca binaria risale al Paragrafo 5.1.3, come strumento per determinare se un determinato oggetto fosse presente all'interno di una sequenza ordinata. Nella nostra prima presentazione (nel Codice 5.3), il metodo `binarySearch`

restituiva `true` o `false` per indicare, rispettivamente, se l'oggetto cercato era stato trovato oppure no.

L'importante modifica di cui parliamo qui consiste nel fatto che, quando si esegue una ricerca per bisezione, si può restituire, invece di un valore booleano, un indice, che segnali la posizione in cui l'oggetto cercato è stato trovato, o una posizione vicina se non è stato trovato. Durante una ricerca che abbia successo, il metodo determina con precisione l'indice in cui ha trovato l'oggetto cercato. Durante una ricerca infruttuosa, nonostante l'oggetto cercato non venga trovato, l'algoritmo determinerà in modo efficiente una coppia di indici che individuano quegli elementi, all'interno del contenitore, che hanno un valore appena inferiore e appena superiore a quello dell'oggetto mancante.

Nel Codice 10.11 e 10.12 presentiamo un'implementazione completa di una classe, `SortedTableMap`, che realizza il tipo di dato astratto "mappa ordinata". La caratteristica più significativa del nostro progetto è la presenza del metodo ausiliario `findIndex`. Questo metodo usa l'algoritmo ricorsivo di ricerca binaria, ma restituisce l'*indice* della voce più a sinistra, nell'intervallo di ricerca, che ha chiave non minore di *k*; se, nell'intervallo di ricerca, nessuna voce contiene una tale chiave, viene restituito l'indice che indica la posizione (inesistente) immediatamente successiva all'ultima dell'intervallo di ricerca. Usando questa convenzione, se una voce contiene la chiave cercata, la ricerca restituisce ovviamente l'indice di tale voce (ricordando che le chiavi sono univoche all'interno della mappa). Se, invece, la chiave cercata è assente, il metodo restituisce l'indice in cui si dovrebbe inserire una nuova voce avente quella chiave.

Codice 10.11: Un'implementazione della classe `SortedTableMap` (che prosegue nel Codice 10.12). La classe di base `AbstractSortedMap` (non riportata nel libro) definisce il metodo ausiliario, `compare`, che usa il comparatore disponibile.

```

1  public class SortedTableMap<K,V> extends AbstractSortedMap<K,V> {
2      private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
3      public SortedTableMap() { super(); }
4      public SortedTableMap(Comparator<K> comp) { super(comp); }
5      /** Restituisce il minimo indice in table[low..high] in cui si trova una voce
6          con chiave non minore di k (altrimenti, per convenzione, high+1). */
7      private int findIndex(K key, int low, int high) {
8          if (high < low) return high + 1;           // nessuna voce da esaminare
9          int mid = (low + high) / 2;
10         int comp = compare(key, table.get(mid));
11         if (comp == 0)
12             return mid;                         // trovata una corrispondenza esatta
13         else if (comp < 0)
14             return findIndex(key, low, mid - 1); // se c'è, è a sinistra di mid
15         else
16             return findIndex(key, mid + 1, high); // se c'è, è a destra di mid
17     }
18     /** Versione di findIndex che cerca nell'intera tabella. */
19     private int findIndex(K key) { return findIndex(key, 0, table.size() - 1); }
20     /** Restituisce il numero di voci nella mappa. */
21     public int size() { return table.size(); }
22     /** Restituisce il valore associato alla chiave specificata (altrimenti null). */
23     public V get(K key) {
24         int j = findIndex(key);
25         if (j == size() || compare(key, table.get(j)) != 0) return null; // non c'è
26         return table.get(j).getValue();

```

```

27 }
28 /** Associa il valore dato alla chiave data, restituendo il vecchio valore. */
29 public V put(K key, V value) {
30     int j = findIndex(key);
31     if (j < size() && compare(key, table.get(j)) == 0) // trovata una corrispondenza
32         return table.get(j).setValue(value);
33     table.add(j, new MapEntry<K,V>(key, value)); // altrimenti inserisce nuova voce
34     return null;
35 }
36 /** Elimina la voce con chiave k (se c'è) e restituisce il valore associato. */
37 public V remove(K key) {
38     int j = findIndex(key);
39     if (j == size() || compare(key, table.get(j)) != 0) return null; // non c'è
40     return table.remove(j).getValue();
41 }

```

Codice 10.12: Un'implementazione della classe SortedTableMap (continua dal Codice 10.11).

```

42 /** Metodo ausiliario, restituisce la voce di indice j, o null se j non valido. */
43 private Entry<K,V> safeEntry(int j) {
44     if (j < 0 || j >= table.size()) return null;
45     return table.get(j);
46 }
47 /** Restituisce la voce con chiave minima (o null se la mappa è vuota). */
48 public Entry<K,V> firstEntry() { return safeEntry(0); }
49 /** Restituisce la voce con chiave massima (o null se la mappa è vuota). */
50 public Entry<K,V> lastEntry() { return safeEntry(table.size()-1); }
51 /** Restituisce la voce con chiave minima tra quelle non minori di key (o null). */
52 public Entry<K,V> ceilingEntry(K key) {
53     return safeEntry(findIndex(key));
54 }
55 /** Restituisce la voce con chiave max tra quelle non maggiori di key (o null). */
56 public Entry<K,V> floorEntry(K key) {
57     int j = findIndex(key);
58     if (j == size() || !key.equals(table.get(j).getKey()))
59         j--; // cerca una posizione più indietro (se non ha trovato la chiave cercata)
60     return safeEntry(j);
61 }
62 /** Restituisce la voce con chiave massima tra quelle minori di key (o null). */
63 public Entry<K,V> lowerEntry(K key) {
64     return safeEntry(findIndex(key) - 1); // proprio a sinistra di ceilingEntry
65 }
66 /** Restituisce la voce con chiave minima tra quelle maggiori di key (o null). */
67 public Entry<K,V> higherEntry(K key) {
68     int j = findIndex(key);
69     if (j < size() && key.equals(table.get(j).getKey()))
70         j++; // a destra della corrispondenza esatta
71     return safeEntry(j);
72 }
73 // nel seguito, supporto per iteratori snapshot per entrySet() e subMap()
74 private Iterable<Entry<K,V>> snapshot(int startIndex, K stop) {
75     ArrayList<Entry<K,V>> buffer = new ArrayList<>();
76     int j = startIndex;
77     while (j < table.size() && (stop == null || compare(stop, table.get(j)) > 0))
78         buffer.add(table.get(j++));
79     return buffer;
80 }

```

```

    public Iterable<Entry<K,V>> entrySet() { return snapshot(o, null); }
    public Iterable<Entry<K,V>> subMap(K fromKey, K toKey) {
        return snapshot(findIndex(fromKey), toKey);
    }
}

```

Analisi

Concludiamo il paragrafo analizzando le prestazioni della nostra implementazione `SortedTableMap`. La Tabella 10.3 riassume i tempi d'esecuzione di tutti i metodi dell'ADT "mappa ordinata" (compresi quelli delle operazioni di una mappa standard). Dovrebbe risultare evidente che i metodi `size`, `firstEntry` e `lastEntry` vengono eseguiti in un tempo $O(1)$ e che la scansione delle chiavi presenti nella tabella può essere eseguita, in una direzione o nell'altra, in un tempo $O(n)$.

Tabella 10.3: Prestazioni di una mappa ordinata implementata da `SortedTableMap`. Usiamo n per indicare il numero di voci nella mappa nel momento in cui viene eseguita l'operazione. Lo spazio occupato in memoria è $O(n)$.

Metodo	Tempo d'esecuzione
<code>size</code>	$O(1)$
<code>get</code>	$O(\log n)$
<code>put</code>	$O(n)$; $O(\log n)$ se la mappa ha una voce con la chiave data
<code>remove</code>	$O(n)$
<code>firstEntry</code> , <code>lastEntry</code>	$O(1)$
<code>ceilingEntry</code> , <code>floorEntry</code> , <code>lowerEntry</code> , <code>higherEntry</code>	$O(\log n)$
<code>subMap</code>	$O(s + \log n)$ quando viene restituito un contenitore con s voci
<code>entrySet</code> , <code>keySet</code> , <code>values</code>	$O(n)$

Il tempo di tutte le diverse forme di ricerca deriva dal fatto che la ricerca binaria in una tabella avente n voci viene eseguita in un tempo $O(\log n)$. Questa affermazione è stata già dimostrata nella Proposizione 5.2, nel Paragrafo 5.2, e quell'analisi si applica chiaramente anche al nostro metodo `findIndex`. Affermiamo, quindi, che il tempo d'esecuzione nel caso peggiore per i metodi `get`, `ceilingEntry`, `floorEntry`, `lowerEntry` e `higherEntry` è $O(\log n)$: ciascuno di questi effettua un'unica invocazione di `findIndex`, seguita da un numero costante di ulteriori passi, necessari per determinare la risposta corretta sulla base dell'indice ricevuto.

L'analisi del metodo `subMap` è un po' più interessante. Il metodo inizia con una ricerca binaria che ha lo scopo di trovare la prima voce appartenente all'intervallo (se ne esiste almeno una). Successivamente, esegue un ciclo, che richiede un tempo $O(1)$ per ogni sua iterazione, con l'obiettivo di individuare i valori successivi che appartengono alla (sotto) mappa, fino a quando raggiunge la fine dell'intervallo di chiavi considerato. Se nell'intervallo sono presenti s chiavi da prendere in esame, il tempo d'esecuzione complessivo è $O(s + \log n)$.

Diversamente dalle operazioni di ricerca, che sono molto efficienti, le operazioni di modifica di una mappa ordinata possono richiedere un tempo considerevole. Anche se la ricerca binaria può essere d'aiuto nell'individuazione dell'indice in cui deve avvenire

la modifica, tanto gli inserimenti quanto le rimozioni richiedono, nel caso peggiore, lo spostamento di un numero lineare di elementi, che devono scorrere di una posizione per preservare l'ordinamento della tabella. Nello specifico, il tempo d'esecuzione $O(n)$ nel caso peggiore è dovuto alle invocazioni di `table.add` all'interno di `put` e di `table.remove` all'interno di `remove` (si veda, nel Paragrafo 7.2, la discussione relativa alle corrispondenti operazioni nella classe `ArrayList`).

In conclusione, le tabelle ordinate vengono utilizzate principalmente in situazioni nelle quali ci si aspetta di dover fare molte ricerche ma relativamente poche modifiche.

10.3.2 Due applicazioni di mappe ordinate

In questo paragrafo analizzeremo applicazioni che traggono particolarmente vantaggio dall'uso di una mappa *ordinata* piuttosto che di una mappa tradizionale, non ordinata. Per poter usare una mappa ordinata, le chiavi devono appartenere a un insieme totalmente ordinato. Inoltre, per trarre vantaggio dalle ricerche effettuate su intervalli di una mappa ordinata, devono esistere dei motivi per cui sia importante cercare nei pressi di una chiave.

Una base di dati per voli aerei

In Internet esistono molti siti web che consentono di fare ricerche su basi di dati per voli aerei, al fine di trovare voli tra diverse città, tipicamente con l'obiettivo di acquistare un biglietto. Per interrogare la base di dati, l'utente specifica le città di partenza e di arrivo, nonché una data e un orario di partenza. Per consentire l'esecuzione di interrogazioni (*query*) di questo tipo, possiamo rappresentare la base di dati con una mappa, dove le chiavi sono oggetti di tipo `Flight` ("volo"), con campi che corrispondono ai quattro parametri citati. In pratica, una chiave è una quadrupla di questo tipo:

$$k = (\text{partenza}, \text{arrivo}, \text{data}, \text{ora})$$

L'oggetto può, poi, memorizzare ulteriori informazioni relative al volo, come il numero identificativo del volo, il numero di posti ancora disponibili in prima classe (F) o in classe turistica (Y), la durata del volo e il prezzo del biglietto.

Il problema di trovare un volo che soddisfi i criteri previsti nell'interrogazione non si risolve cercando un dato preciso corrispondente alla richiesta. Anche se tipicamente l'utente vuole trovare una corrispondenza esatta per le città di partenza e di arrivo, potrebbe avere una certa flessibilità sulla data di partenza e quasi certamente flessibile è l'orario di partenza. Possiamo gestire interrogazioni di questo tipo ordinando le chiavi in senso lessicografico, poi, un'implementazione efficiente di mappa ordinata potrà soddisfare in modo adeguato le richieste degli utenti. Ad esempio, data la chiave k corrispondente a un'interrogazione, potremmo invocare `ceilingEntry(k)` e restituire il primo volo avente le città di partenza e arrivo richieste, con data e orario di partenza che corrispondono a quanto richiesto o sono immediatamente successive. Ancora meglio, se progettiamo bene le chiavi, potremmo usare `subMap(k_1 , k_2)` per trovare tutti i voli che appartengono a un determinato intervallo orario. Ad esempio, se $k_1 = (\text{ORD}, \text{PVD}, \text{05May}, 09:30)$ e $k_2 = (\text{ORD}, \text{PVD}, \text{05May}, 20:00)$, l'invocazione `subMap(k_1 , k_2)` potrebbe generare la seguente sequenza di coppie chiave-valore:

$$(\text{ORD}, \text{PVD}, \text{05May}, 09:53) : (\text{AA 1840}, \text{F5}, \text{Y15}, \text{02:05}, \$251)$$

(ORD, PVD, 05May, 13:29) : (AA 600, F2, Y0, 02:16, \$713)
 (ORD, PVD, 05May, 17:39) : (AA 416, F3, Y9, 02:09, \$365)
 (ORD, PVD, 05May, 19:50) : (AA 1828, F9, Y25, 02:13, \$186)

Insieme dei massimi (*maxima set*)

La vita è un continuo compromesso e spesso dobbiamo trovare un compromesso tra il valore di prestazione desiderato e il costo corrispondente. Supponiamo, ad esempio, di essere interessati alla gestione di una base di dati di valutazioni di automobili in funzione della loro velocità massima e del loro costo. Ci piacerebbe poter consentire a un acquirente che dispone di una certa somma di denaro di interrogare la base di dati per trovare la macchina più veloce tra quelle che si può permettere di acquistare.

Si può progettare un modello per un problema di questo tipo, relativo alla ricerca di un compromesso, usando una coppia chiave-valore per rappresentare i due parametri sui quali cerchiamo il compromesso: in questo caso, sarebbero la coppia costo-velocità, (*cost, speed*), per ciascuna automobile. Osserviamo che, usando questo criterio, alcune automobili sono definitivamente migliori di altre: ad esempio, un'automobile la cui coppia costo-velocità sia (30000, 100) è definitivamente migliore di un'automobile la cui coppia sia (40000, 90). Al tempo stesso, ci sono automobili che non sono "dominate" da altre: ad esempio, come si può vedere nella Figura 10.9, l'automobile (30000, 100) può essere migliore o peggiore dell'automobile (40000, 120), in relazione al denaro disponibile.

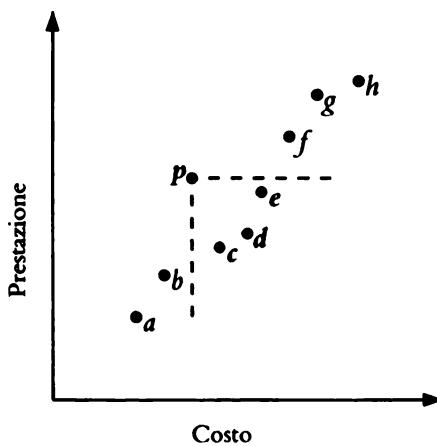


Figura 10.9: Visualizzazione del compromesso tra costo e prestazione, con le coppie rappresentate da punti nel piano. Si noti che il punto p è definitivamente migliore dei punti c, d ed e , ma può essere migliore o peggiore dei punti a, b, f, g e h , in relazione al prezzo che si vuole pagare. Di conseguenza, se aggiungessimo p al nostro insieme, potremmo eliminare i punti c, d e e , ma non gli altri.

Dal punto di vista formale, diciamo che una coppia costo-prestazione, (a, b) , *domina* la coppia $(c, d) \neq (a, b)$ se $a \leq c$ e $b \geq d$, cioè se la prima coppia non ha un costo maggiore e ha prestazioni almeno pari. Una coppia (a, b) è un *massimo* se non è dominata da nessun'altra coppia. Vogliamo cercare di gestire l'insieme dei massimi di un contenitore di coppie costo-

prestazione, cioè vogliamo aggiungere nuove coppie a questo contenitore (ad esempio, quando si rende disponibile una nuova automobile) e interrogare il contenitore stesso per una data somma di denaro, d , in modo che ci restituisca l'automobile più veloce che non costi più di d dollari.

Gestire un insieme dei massimi usando una mappa ordinata

Possiamo memorizzare l'insieme delle coppie massime in una mappa ordinata, in modo che il costo sia la chiave e le prestazioni (in questo caso, la velocità) sia il valore. Possiamo, poi, implementare le operazioni $\text{add}(c, p)$, che aggiunge una nuova coppia costo-prestazione (c, p) , e $\text{best}(c)$, che restituisce la coppia che ha la prestazione migliore tra quelle con costo massimo c . Il Codice 10.13 mostra un'implementazione di tale classe, che abbiamo chiamato `CostPerformanceDatabase`.

Codice 10.13: Un'implementazione di una classe che gestisce un insieme di voci massime di tipo costo-prestazione usando una mappa ordinata.

```

1  /** Gestisce una base di dati di coppie massime (costo, prestazione). */
2  public class CostPerformanceDatabase {
3
4      SortedMap<Integer, Integer> map = new SortedTableMap<>();
5
6      /** Costruisce una base di dati inizialmente vuota. */
7      public CostPerformanceDatabase() { }
8
9      /** Restituisce la voce (costo, prestazione) avente prestazione massima con
10       * costo che non supera c (oppure null se nessuna voce ha costo inferiore a c).
11       */
12     public Entry<Integer, Integer> best(int cost) {
13         return map.floorEntry(cost);
14     }
15
16     /** Aggiunge una nuova voce con costo c e prestazione p. */
17     public void add(int c, int p) {
18         Entry<Integer, Integer> other = map.floorEntry(c); // costo di other <= c
19         if (other != null && other.getValue() >= p) // se velocità di other >= p
20             return; // (c,p) è dominata, quindi va ignorata
21         map.put(c, p); // altrimenti, aggiunge (c,p) al database
22         // e ora elimina tutte le voci che sono dominate dalla nuova coppia
23         other = map.higherEntry(c); // costo di other > c
24         while (other != null && other.getValue() <= p) { // se velocità non > p
25             map.remove(other.getKey()); // elimina la voce other
26             other = map.higherEntry(c);
27         }
28     }
29 }
```

Sfortunatamente, se implementiamo la mappa ordinata usando la classe `SortedTableMap`, il metodo `add` ha un tempo d'esecuzione $O(n)$ nel caso peggiore. Se, d'altra parte, implementiamo la mappa usando una *skip list*, che descriveremo nel prossimo capitolo, possiamo eseguire le interrogazioni con `best(c)` in un tempo atteso $O(\log n)$ e gli inserimenti con `add(c, p)` in un tempo atteso $O((1 + r) \log n)$, dove r è il numero di voci eliminate.

10.4 Skip list

Nel Paragrafo 10.3.1 abbiamo detto che una tabella ordinata consente di effettuare ricerche in un tempo $O(\log n)$ usando l'algoritmo di ricerca binaria. Sfortunatamente, però, le operazioni di aggiornamento di una tabella ordinata richiedono un tempo d'esecuzione $O(n)$ nel caso peggiore, per la necessità di far scorrere gli elementi. Nel Capitolo 7 abbiamo dimostrato che le liste concatenate consentono di eseguire in modo molto efficiente le operazioni di aggiornamento, a patto che sia stata individuata in precedenza la posizione all'interno della lista. Sfortunatamente, non possiamo però eseguire ricerche veloci in una lista concatenata normale: ad esempio, l'algoritmo di ricerca binaria richiede uno strumento efficiente per accedere in modo diretto a un elemento della sequenza usando un indice.

Una struttura dati interessante per realizzare in modo efficiente il tipo di dato astratto "mappa ordinata" è la *skip list* ("lista con salti"). La skip list costituisce un intelligente compromesso per effettuare in modo efficiente sia le ricerche sia le modifiche, e nella libreria standard di Java ne troviamo un'implementazione nella classe `java.util.concurrent.ConcurrentSkipListMap`. La skip list S per una mappa M è costituita da una serie di liste $\{S_0, S_1, \dots, S_h\}$; ciascuna lista S_i memorizza un sottoinsieme delle voci di M ordinato per chiave crescente, con l'aggiunta di voci aventi le due chiavi-sentinella $-\infty$ e $+\infty$, dove $-\infty$ è minore di qualunque chiave che possa essere inserita in M e $+\infty$ è maggiore di qualunque chiave che possa essere inserita in M . Inoltre, le liste di S soddisfano le seguenti proprietà:

- La lista S_0 contiene tutte le voci della mappa M (e le sentinelle $-\infty$ e $+\infty$).
- Per $i = 1, \dots, h - 1$, la lista S_i contiene (oltre alle sentinelle $-\infty$ e $+\infty$) un sottoinsieme casuale delle voci appartenenti alla lista S_{i-1} .
- La lista S_h contiene soltanto le sentinelle $-\infty$ e $+\infty$.

La Figura 10.10 mostra un esempio di skip list. Di solito una skip list S viene visualizzata con la lista S_0 in basso e le liste S_1, \dots, S_h al di sopra di essa. Inoltre, diciamo che h è l'*altezza* (*height*) di S .

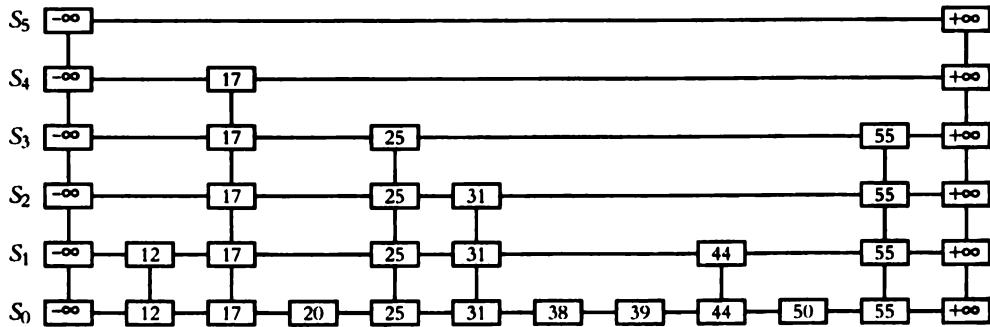


Figura 10.10: Esempio di skip list che memorizza 10 voci. Per semplicità abbiamo riportato soltanto le chiavi delle voci, senza i valori corrispondenti.

Dal punto di vista intuitivo, le liste sono composte in modo che la lista S_{i+1} contenga le voci di S_i , scelte alternativamente, una sì e una no. Tuttavia, il dimezzamento del numero di

voci nel passaggio da una lista alla successiva non è richiesto esplicitamente come proprietà delle skip list: si usa, invece, una selezione casuale. Come vedremo analizzando il metodo di inserimento in dettaglio, le voci di S_{i+1} sono scelte a caso tra le voci di S_i , assegnando ciascuna voce di S_i a S_{i+1} con probabilità $1/2$: si tratta, in pratica, di "lanciare una moneta" per ogni voce di S_i , inserendola in S_{i+1} soltanto se esce "testa". Di conseguenza, ci aspettiamo che S_i abbia circa $n/2$ voci, che S_2 abbia circa $n/4$ voci e che, in generale, S_i abbia circa $n/2^i$ voci. Quindi, ci aspettiamo che l'altezza h di S sia circa $\log n$.

Nei moderni calcolatori sono normalmente presenti funzioni che generano numeri che sembrano casuali, perché vengono usate in modo intenso nei giochi e nelle simulazioni al calcolatore, oltre che nella crittografia. Alcune funzioni, dette *generatori di numeri pseudocasuali*, generano tali numeri a partire da un *seme* (*seed*) iniziale, come visto nella discussione relativa alla classe `java.util.Random` nel Capitolo 3.1.3. Altri metodi usano dispositivi hardware per estrarre dal mondo reale "veri" numeri casuali. In ogni caso, nella nostra analisi ipotizzeremo che i calcolatori abbiano accesso a numeri sufficientemente casuali.

Usando la *casualità* nella progettazione di strutture dati e algoritmi si ottiene il vantaggio di poter disporre di metodi e strutture dati piuttosto efficienti e semplici. La ricerca in una skip list ha le stesse prestazioni temporali logaritmiche dell'algoritmo di ricerca binaria, ma estende tali prestazioni anche ai metodi di aggiornamento che effettuano inserimenti o rimozioni. Ciò nonostante, per la skip list i valori limite delle prestazioni sono *attesi* (*expected*), in senso probabilistico, mentre nella ricerca binaria all'interno di una tabella ordinata tale valore limite è relativo al *caso peggiore*.

Una skip list compie scelte casuali nel disporre le voci all'interno della struttura, in modo che le ricerche e le modifiche richiedano *in media* un tempo $O(\log n)$, dove n è il numero di voci presenti nella mappa. È interessante osservare come il concetto di complessità temporale media usato in questo contesto non dipenda dalla distribuzione di probabilità delle chiavi: dipende, invece, soltanto dal generatore di numeri pseudocasuali usato, nell'implementazione del metodo di inserimento, per decidere dove posizionare la nuova voce. Il tempo d'esecuzione viene mediato su tutti i possibili numeri casuali utilizzati per inserire voci.

Come nell'astrazione di posizione utilizzata per le liste e per gli alberi, possiamo vedere una skip list come un contenitore bidimensionale di posizioni, disposte orizzontalmente in *livelli* e verticalmente in *torri*. Ogni livello è una lista S_i e ogni torre contiene posizioni che memorizzano la stessa voce in liste consecutive. Le posizioni presenti in una skip list possono essere scandite usando le seguenti operazioni:

- next(*p*):** Restituisce la posizione che segue *p* sullo stesso livello.
- prev(*p*):** Restituisce la posizione che precede *p* sullo stesso livello.
- above(*p*):** Restituisce la posizione che si trova sopra a *p* nella stessa torre.
- below(*p*):** Restituisce la posizione che si trova sotto a *p* nella stessa torre.

Tutte queste operazioni restituiscono convenzionalmente `null` se la posizione richiesta non esiste. Senza entrare nei dettagli, osserviamo che possiamo implementare facilmente una skip list mediante una struttura concatenata, in modo che i singoli metodi di scansione

richiedano un tempo $O(1)$, data una posizione p della skip list. Tale struttura concatenata è essenzialmente un contenitore di h liste doppiamente concatenate allineate alle torri, che sono anch'esse liste doppiamente concatenate.

10.4.1 Operazioni di ricerca e modifica in una skip list

La struttura di una skip list consente di utilizzare algoritmi semplici per eseguire ricerche nella mappa e modificarla. Infatti, tutte tali operazioni sono basate su un unico elegante algoritmo, `SkipSearch`, che riceve una chiave k e trova nella lista S_0 la posizione p della voce che ha la chiave massima (che può essere anche $-\infty$) tra quelle non maggiori di k .

Ricerca in una skip list

Supponiamo di voler cercare una chiave k . Iniziamo l'algoritmo `SkipSearch` assegnando alla variabile p la posizione più in alto e più a sinistra della skip list S , la cosiddetta *posizione iniziale (start position)* di S . La posizione iniziale di S è, quindi, la posizione di S_h che contiene la voce sentinella, con chiave $-\infty$. Poi, eseguiamo i seguenti passi (visibili nella Figura 10.11), dove $\text{key}(p)$ indica la chiave della voce che si trova nella posizione p .

1. Se $S.\text{below}(p)$ è null, la ricerca termina: abbiamo raggiunto il fondo e abbiamo individuato in S la voce avente chiave massima tra quelle non maggiori di k . Altrimenti, scendiamo di un livello nella torre in cui ci troviamo, ponendo $p = S.\text{below}(p)$.
2. A partire dalla posizione p , spostiamo p in avanti (cioè verso destra) finché non si porta, nel livello in cui ci troviamo, nella posizione più a destra che abbia $\text{key}(p) \leq k$ (chiamiamo questa fase *scansione in avanti*). Osserviamo che tale posizione esiste sempre, perché tutti i livelli contengono le chiavi $-\infty$ e $+\infty$. Dopo aver eseguito questa scansione in avanti lungo il livello in cui si trova la posizione p , può darsi che p sia rimasta dov'era inizialmente.
3. Torniamo al passo 1.

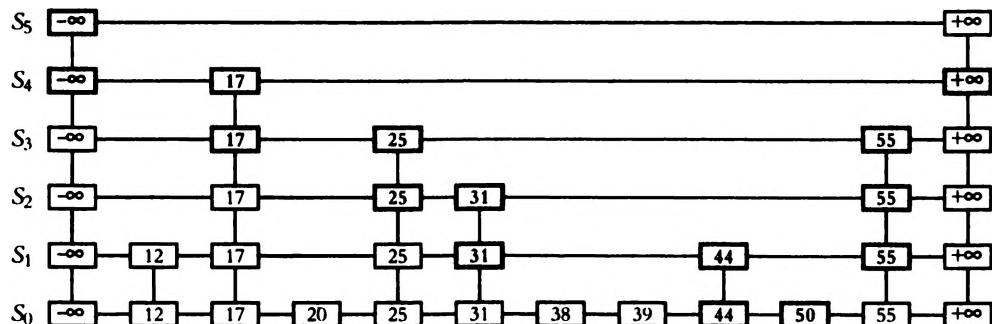


Figura 10.11: Esempio di ricerca in una skip list. Le posizioni evidenziate sono quelle che vengono esaminate quando si cerca la chiave 50.

Nel Codice 10.14 proponiamo una descrizione mediante pseudocodice dell'algoritmo di ricerca in una skip list, `SkipSearch`. Usando questo metodo, possiamo eseguire l'operazione `get(k)` sulla mappa calcolando $p = \text{SkipSearch}(k)$ e controllando se $\text{key}(p) = k$ oppure

Se le due chiavi confrontate sono uguali restituiamo il valore associato, altrimenti restituiamo null.

Codice 10.14: Algoritmo che cerca la chiave k nella skip list S . La variabile s contiene la posizione iniziale di S .

Algoritmo SkipSearch(k):

Input: Una chiave k da cercare

Output: La posizione p nella lista S_0 , avente la chiave massima tra quelle non maggiori di k

```

 $p = s$                                 { parte dalla posizione iniziale }
while  $\text{below}(p) \neq \text{null}$  do
     $p = \text{below}(p)$                       { scende }
    while  $k \geq \text{key}(\text{next}(p))$  do
         $p = \text{next}(p)$                       { avanza }
    return  $p$ 

```

Si può dimostrare che il tempo d'esecuzione atteso per l'algoritmo `SkipSearch` applicato a una skip list avente n voci è $O(\log n)$, tuttavia dimostreremo questa affermazione più avanti, dopo aver parlato dell'implementazione dei metodi di aggiornamento della skip list. La navigazione che parte dalla posizione identificata da `SkipSearch(k)` può essere facilmente utilizzata anche per realizzare le altre forme di ricerca tipiche del tipo di dato astratto "mappa ordinata", come `ceilingEntry` o `subMap`.

Inserimento in una skip list

L'esecuzione dell'operazione `put(k, v)`, tipica della mappa, inizia con un'invocazione di `SkipSearch(k)`, che restituisce la posizione p della voce del livello più basso che ha la chiave massima tra quelle che hanno chiave non maggiore di k (osservando che p può anche fare riferimento alla voce speciale, contenente la chiave-sentinella $-\infty$). Se $\text{key}(p) = k$, il valore associato a k viene sovrascritto con v , altrimenti dobbiamo creare una nuova torre per la voce (k, v) . Inseriamo (k, v) subito dopo la posizione p all'interno di S_0 . Dopo aver inserito la nuova voce al livello più basso, usiamo una scelta casuale per decidere l'altezza della torre relativa a tale nuova voce: "lanciamo una moneta" e, se viene "croce", ci fermiamo; altrimenti (cioè se viene "testa") risaliamo al livello immediatamente superiore e inseriamo (k, v) anche in tale livello, nella posizione corretta, quindi lanciamo di nuovo una moneta e ripetiamo la procedura. Di conseguenza, continuiamo a inserire la nuova voce (k, v) nelle liste finché non facciamo un lancio il cui risultato sia "testa". Creiamo la torre collegando insieme tutti i riferimenti alla nuova voce (k, v) che abbiamo creato durante l'esecuzione di questa procedura. Un lancio di moneta sufficientemente equo si può ottenere usando il generatore di numeri casuali predefinito nella libreria standard di Java, creando un esemplare di `java.util.Random` e invocando il suo metodo `nextBoolean()`, che restituisce `true` o `false` con probabilità $1/2$ per ciascuno dei due eventi.

L'algoritmo che effettua un inserimento in una skip list S è riportato nel Codice 10.15 e illustrato nella Figura 10.12: usa un'invocazione del metodo `insertAfterAbove($p, q, (k, v)$)` che inserisce dopo la posizione p (sullo stesso livello di p) e sopra la posizione q una nuova posizione, r , contenente la voce (k, v) , restituendo proprio r , dopo aver sistemato i riferimenti interni alle posizioni p , q e r in modo che i metodi `next`, `prev`, `above` e `below` funzionino

correttamente. Il tempo d'esecuzione atteso dell'algoritmo di inserimento in una skip list avente n voci è $O(\log n)$, come dimostreremo nel Paragrafo 10.4.2.

Codice 10.15: Inserimento in una skip list della voce (k, v) , nell'ipotesi che la struttura non contenga una voce con chiave k . Il metodo coinFlip() restituisce "heads" (testa) o "tails" (croce) con probabilità 1/2 per ciascuno dei due eventi. Le variabili di esemplare n, h, s contengono, rispettivamente, il numero di voci, l'altezza della skip list e la sua posizione iniziale.

Algoritmo SkipInsert(k, v):

Input: Una chiave k e un valore v

Output: La posizione più alta tra quelle occupate dalla nuova voce inserita nella skip list

$p = \text{SkipSearch}(k)$ { posizione in S_h della chiave massima non maggiore di k }

$q = \text{null}$ { posizione corrente nella torre della nuova voce }

$i = -1$ { altezza corrente della torre della nuova voce }

repeat

$i = i + 1$ { incrementa l'altezza della torre della nuova voce }

if $i \geq h$ then

$h = h + 1$ { aggiunge un nuovo livello alla skip list }

$t = \text{next}(s)$

$s = \text{insertAfterAbove}(\text{null}, s, (-\infty, \text{null}))$ { alza la torre più a sinistra }

$\text{insertAfterAbove}(s, t, (+\infty, \text{null}))$ { alza la torre più a destra }

$q = \text{insertAfterAbove}(p, q, (k, v))$ { alza la torre della nuova voce }

while $\text{above}(p) == \text{null}$ do

$p = \text{prev}(p)$ { retrocede orizzontalmente }

$p = \text{above}(p)$ { sale di un livello }

until $\text{coinFlip()} == \text{tails}$

$n = n + 1$

return q { posizione più in alto nella torre della nuova voce }

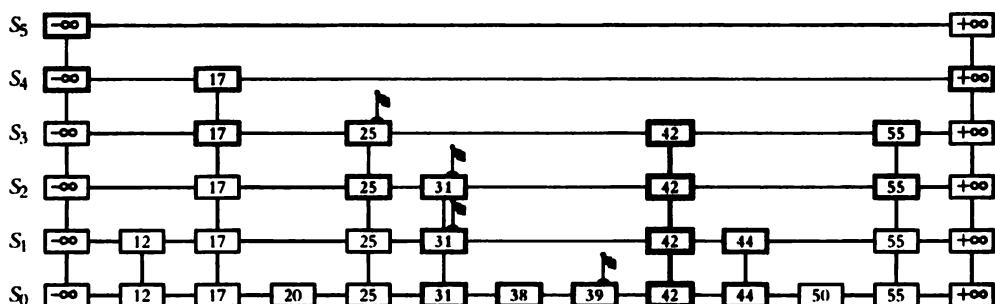


Figura 10.12: Inserimento di una nuova voce con chiave 42 nella skip list della Figura 10.10 usando il metodo SkipInsert (descritto nel Codice 10.15). Ipotizziamo che i "lanci di moneta" casuali che servono per inserire la nuova voce producano una sequenza di tre "testa" consecutivi, seguiti da "croce". Le posizioni visitate sono evidenziate con un tratto più spesso. Le posizioni della torre della nuova voce (variabile q) sono disegnate con un tratto ancora più spesso, mentre sulle posizioni che le precedono (variabile p) è stata posta una piccola bandiera.

Rimozione da una skip list

Come gli algoritmi di ricerca e di inserimento, l'algoritmo di rimozione di una voce da una skip list è abbastanza semplice e, in effetti, è anche più semplice dell'algoritmo di inserimento. Per eseguire l'operazione $\text{remove}(k)$, tipica di una mappa, cominceremo eseguendo il metodo $\text{SkipSearch}(k)$. Se la posizione p restituita contiene una voce con chiave diversa da k , restituiamoci `null`, perché non c'è niente da rimuovere; altrimenti, eliminiamo p e tutte le posizioni al di sopra di p , a cui si accede facilmente usando ripetutamente l'operazione `above` per "arrampicarsi" in S sulla torre della voce rimossa, a partire da p . Eliminando uno dopo l'altro i livelli della torre, sistemiamo i collegamenti tra le posizioni orizzontalmente adiacenti a ciascuna posizione rimossa. L'algoritmo di rimozione è illustrato nella Figura 10.13 e la sua descrizione dettagliata mediante pseudocodice è lasciata come esercizio (R-10.24). Come mostriremo nel Paragrafo 10.4.2, il tempo d'esecuzione atteso per l'operazione `remove` in una skip list con n voci è $O(\log n)$.

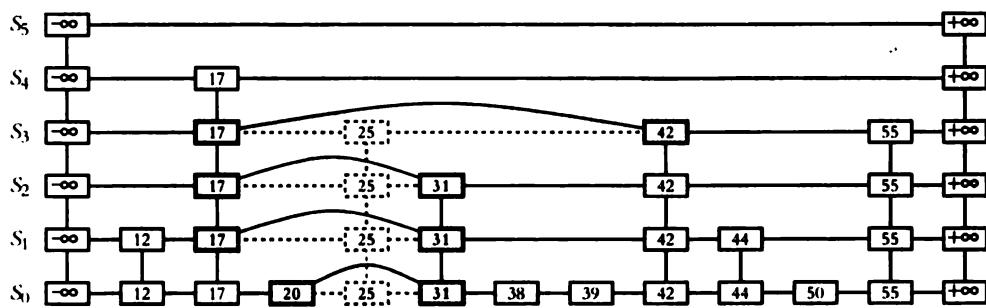


Figura 10.13: Eliminazione della voce con chiave 25 dalla skip list della Figura 10.12. Le posizioni evidenziate sono quelle che vengono visitate dopo la ricerca della posizione di S_0 che contiene la voce da eliminare. Le posizioni rimosse sono tratteggiate.

Prima di fare questa analisi, però, vogliamo discutere alcuni miglioramenti di minore entità che si possono apportare alla struttura dati "skip list". Innanzitutto, non c'è alcun bisogno che, al di sopra del livello più basso, le voci contengano effettivamente riferimenti ai valori, tutto ciò che serve a quei livelli sono i riferimenti alle chiavi. In effetti, una rappresentazione più efficiente della torre prevede che questa sia un unico oggetto, che memorizza un'unica coppia chiave-valore e, se la torre raggiunge il livello S_j , gestisce j riferimenti all'indietro e j riferimenti in avanti. In secondo luogo, per quanto riguarda gli assi orizzontali della struttura, è possibile che tali liste siano a concatenazione singola, anziché doppia, memorizzando soltanto i riferimenti in avanti, perché gli inserimenti e le rimozioni possono essere eseguiti procedendo soltanto dall'alto verso il basso e da sinistra verso destra: analizzeremo i dettagli di questa seconda ottimizzazione nell'Esercizio C-10.55. Nessuna di queste due ottimizzazioni migliora le prestazioni temporali asintotiche della skip list per più di un fattore costante, ma possono, comunque, essere sensate, dal punto di vista pratico. In effetti, le evidenze sperimentali suggeriscono che le skip list così ottimizzate siano più veloci degli alberi AVL e di altri alberi di ricerca bilanciati che vedremo nel Capitolo 11.

Gestione del livello più alto

Una skip list S deve conservare in una variabile di esemplare il riferimento alla propria posizione iniziale (che, ricordiamo, è la posizione più in alto e più a sinistra in S) e deve avere una politica di gestione di eventuali inserimenti che cerchino di continuare a far crescere la torre di una nuova voce oltre il livello più alto di S . Si possono utilizzare due diverse strategie, ciascuna con i propri vantaggi.

Una possibilità è quella di fare in modo che l'altezza della skip list, h , abbia un valore prefissato in funzione di n , cioè del numero di voci presenti nella mappa (facendo l'analisi delle prestazioni vedremo che $h = \max\{10, 2\lceil \log n \rceil\}$ è una scelta ragionevole e $h = 3\lceil \log n \rceil$ è ancora più sicuro). Per implementare questa strategia bisogna modificare l'algoritmo di inserimento in modo che interrompa la procedura che fa crescere la torre di una nuova voce quando questa ha raggiunto il livello massimo (a meno che non sia $\lceil \log n \rceil < \lceil \log(n+1) \rceil$, nel qual caso si può procedere per un ulteriore livello, perché il valore limite di h sta per aumentare).

L'altra possibilità è lasciare che, durante l'inserimento, la nuova torre continui a crescere fintanto che il generatore di numeri casuali continua a restituire "testa". Questo è, in effetti, l'approccio scelto dall'algoritmo `SkipInsert` riportato nel Codice 10.15. Come metteremo in evidenza nell'analisi delle prestazioni delle skip list, la probabilità che un inserimento proceda fino a un livello che superi $O(\log n)$ è molto bassa, per cui anche questa scelta di progetto dovrebbe funzionare bene.

Entrambe le strategie producono un tempo d'esecuzione atteso pari a $O(\log n)$ per la ricerca, l'inserimento e la rimozione, come vedremo nel prossimo paragrafo.

10.4.2 Analisi probabilistica delle prestazioni di una skip list*

Come abbiamo appena scritto, la skip list consente di implementare in modo semplice una mappa ordinata, ma, in termini di prestazioni di caso peggiore, non è una struttura particolarmente interessante. In effetti, se non viene preclusa la possibilità che un inserimento continui a far crescere la nuova torre significativamente al di sopra dell'attuale livello più elevato della struttura, l'algoritmo di inserimento può entrare in quello che può rivelarsi un ciclo quasi infinito (non è un vero e proprio ciclo infinito, perché la probabilità che una moneta, lanciata ripetutamente, continui a presentare la "testa" per sempre è zero). Inoltre, non possiamo aggiungere indefinitamente posizioni a una lista senza, prima o poi, esaurire la memoria disponibile. In ogni caso, se terminiamo la procedura di inserimento di una nuova voce quando viene raggiunto il livello più elevato, h , allora il tempo d'esecuzione *nel caso peggiore* per le operazioni `get`, `put` e `remove` di una skip list S avente n voci e altezza h è $O(n + h)$. Queste prestazioni del caso peggiore si verificano quando la torre di ogni nuova voce inserita raggiunge il livello $h - 1$, essendo h , come al solito, l'altezza di S . Questo evento, però, ha una probabilità estremamente bassa. Sulla base di questo caso peggiore, potremmo concludere che la struttura *skip list* sia definitivamente peggiore delle altre implementazioni di mappa discusse nei primi paragrafi di questo capitolo, ma questa non sarebbe un'analisi equa, perché questo comportamento di caso peggiore è grossolanamente sovrastimato.

Un limite superiore per l'altezza di una skip list

Dato che la fase di inserimento prevede casualità, un'analisi più accurata delle skip list richiede l'uso di un po' di probabilità. A prima vista questo potrebbe sembrare un compito decisamente ostico, perché un'analisi probabilistica approfondita e completa richiederebbe molta matematica (e, in verità, negli articoli più recenti apparsi in letteratura relativamente alle strutture dati queste analisi matematicamente approfondite sono sempre più frequenti). Fortunatamente, però, un tale tipo di analisi non è necessario per comprendere il comportamento asintotico atteso per le operazioni di una skip list. L'analisi probabilistica informale e intuitiva che riportiamo qui usa solamente i concetti basilari della teoria della probabilità.

Iniziamo determinando il valore atteso dell'altezza h di una skip list S con n voci (nell'ipotesi che non si interrompa l'inserimento prematuramente). La probabilità che una determinata voce abbia una torre di altezza $i \geq 1$ è uguale alla probabilità di ottenere i risultati "testa" consecutivi lanciando una moneta e tale probabilità è, come noto, $1/2^i$. Quindi, la probabilità P_i che il livello i abbia almeno una posizione è così limitata:

$$P_i \leq \frac{n}{2^i}$$

perché la probabilità che uno qualsiasi degli n eventi distinti si verifichi è al massimo uguale alla somma delle probabilità che ciascuno di questi accada.

La probabilità che l'altezza h di S sia maggiore di i è uguale alla probabilità che il livello i abbia almeno una posizione, cioè è non maggiore di P_i . Questo significa che, ad esempio, h è maggiore di $3\log n$ con una probabilità che al massimo vale:

$$\begin{aligned} P_{3\log n} &\leq \frac{n}{2^{3\log n}} \\ &= \frac{n}{n^3} = \frac{1}{n^2} \end{aligned}$$

Ad esempio, se $n = 1000$, questa probabilità è uno su un milione. Più in generale, data una costante $c > 1$, h è maggiore di $c \log n$ con una probabilità che, al massimo, vale $1/n^{c-1}$. Quindi, la probabilità che h sia minore di $c \log n$ è almeno $1 - 1/n^{c-1}$. Allora è vero che, con elevata probabilità, l'altezza h di S è $O(\log n)$.

Analisi del tempo d'esecuzione di una ricerca in una skip list

Consideriamo, ora, il tempo d'esecuzione di una ricerca nella skip list S , ricordando che tale ricerca richiede l'esecuzione di due cicli `while` annidati. Il ciclo più interno esegue una scansione in avanti su un livello di S finché la chiave in esame non è maggiore della chiave cercata, k , mentre il ciclo più esterno scende di livello in livello e ripete la scansione in avanti a ogni sua iterazione. Dato che con elevata probabilità l'altezza h di S è $O(\log n)$, il numero di discese è $O(\log n)$ con elevata probabilità.

Ci rimane, quindi, da trovare un limite superiore per il numero di passi in avanti che vengono eseguiti nelle varie scansioni orizzontali. Sia n_i il numero di chiavi esaminate du-

rante la scansione del livello i . Osserviamo che, dopo la chiave che si trova nella posizione iniziale della skip list, ogni ulteriore chiave esaminata in una scansione in avanti al livello i non può appartenere anche al livello $i + 1$, altrimenti l'avremmo incontrata durante la scansione del livello precedente, che, appunto, è il livello $i + 1$. Quindi, la probabilità che una chiave qualsiasi venga considerata nel conteggio n_i è $1/2$, per cui il valore atteso di n_i è esattamente uguale al numero atteso di lanci che occorre fare con una moneta equa prima che esca "testa": questo valore è 2. Di conseguenza, la quantità attesa di tempo dedicato alla scansione in avanti a un livello i è $O(1)$. Dato che S ha, con elevata probabilità, un numero di livelli $O(\log n)$, il valore atteso del tempo d'esecuzione di una ricerca in S è $O(\log n)$. Con un'analisi simile, potremmo dimostrare che il valore atteso del tempo d'esecuzione di un'operazione di inserimento o di rimozione è $O(\log n)$.

Utilizzo dello spazio in una skip list

Infine, dedichiamoci all'occupazione di spazio in memoria di una skip list S avente n voci. Come abbiamo già avuto modo di osservare, il numero atteso di posizioni al livello i è $n/2^i$ e questo significa che il numero totale di posizioni che ci attendiamo di avere in S è:

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^j}$$

Usando la Proposizione 4.5 relativa alle sommatorie geometriche, otteniamo:

$$\sum_{i=0}^h \frac{1}{2^i} = \frac{\left(\frac{1}{2}\right)^{h+1} - 1}{\frac{1}{2} - 1} = 2 \cdot \left(1 - \frac{1}{2^{h+1}}\right) < 2 \quad \text{per ogni } h \geq 0$$

Quindi, il valore atteso dello spazio occupato da S è $O(n)$.

La Tabella 10.4 riassume le prestazioni di una mappa ordinata realizzata con una skip list.

Tabella 10.4: Prestazioni di una mappa ordinata implementata con una skip list. Usiamo n per indicare il numero di voci nella mappa nel momento in cui viene eseguita l'operazione. Lo spazio occupato in memoria è $O(n)$.

Metodo	Tempo d'esecuzione
<code>size, isEmpty</code>	$O(1)$
<code>get</code>	$O(\log n)$ atteso
<code>put</code>	$O(\log n)$ atteso
<code>remove</code>	$O(\log n)$ atteso
<code>firstEntry, lastEntry</code>	$O(1)$
<code>ceilingEntry, floorEntry</code>	$O(\log n)$ atteso
<code>lowerEntry, higherEntry</code>	
<code>subMap</code>	$O(s + \log n)$ atteso, con s voci
<code>entrySet, keySet, values</code>	$O(n)$

10.5 Insiemi, multi-insiemi e multi-mappe

Concludiamo questo capitolo prendendo in esame alcune ulteriori astrazioni che sono strettamente correlate al tipo di dato astratto "mappa" e che si possono implementare usando strutture dati simili a quelle viste per la mappa.

- Un **insieme** (*set*) è una raccolta non ordinata di elementi, priva di duplicati, che tipicamente consente di eseguire in modo efficiente l'operazione di verifica di appartenenza (*membership test*).
- Un **multi-insieme** (*multiset*, chiamato anche *bag*, cioè "sacco" o "borsa") è un contenitore, simile a un insieme, che consente la presenza di elementi duplicati al proprio interno.
- Una **multi-mappa** (*multimap*) è simile a una mappa tradizionale, perché associa valori a chiavi, ma in una multi-mappa una chiave può essere associata a più valori diversi. Ad esempio, l'indice analitico di un libro mette in corrispondenza un determinato termine con una o più pagine all'interno del libro nelle quali compare.

10.5.1 L'insieme come tipo di dato astratto

Il Java Collections Framework definisce l'interfaccia `java.util.Set`, che contiene i seguenti metodi fondamentali:

- `add(e)`: Aggiunge l'elemento *e* a *S* (se non è già presente in *S*).
- `remove(e)`: Eliminare l'elemento *e* da *S* (se è presente in *S*).
- `contains(e)`: Restituisce `true` se e solo se *e* è un elemento di *S*.
- `iterator()`: Restituisce un iteratore per gli elementi di *S*.

È presente anche il supporto per le tradizionali operazioni matematiche tra due insiemi *S* e *T*: **unione** (*union*), **intersezione** (*intersection*) e **sottrazione** (*subtraction*).

$$S \cup T = \{e : e \text{ appartiene a } S \text{ oppure } e \text{ appartiene a } T\}$$

$$S \cap T = \{e : e \text{ appartiene a } S \text{ e } e \text{ appartiene a } T\}$$

$$S - T = \{e : e \text{ appartiene a } S \text{ e } e \text{ non appartiene a } T\}$$

Nell'interfaccia `java.util.Set`, queste operazioni sono realizzate mediante i metodi seguenti, quando eseguiti su un insieme *S*:

- `addAll(T)`: Modifica *S* in modo che contenga anche tutti gli elementi dell'insieme *T*, in pratica sostituendo *S* con $S \cup T$.
- `retainAll(T)`: Modifica *S* in modo che contenga soltanto quegli elementi che sono anche elementi dell'insieme *T*, in pratica sostituendo *S* con $S \cap T$.
- `removeAll(T)`: Modifica *S* rimuovendo tutti quegli elementi che sono anche elementi dell'insieme *T*, in pratica sostituendo *S* con $S - T$.

Possiamo applicare lo schema di progettazione mediante “modello di metodo” (*template method pattern*) per implementare i tre metodi `addAll`, `retainAll` e `removeAll` usando soltanto invocazioni dei metodi fondamentali `add`, `remove`, `contains` e `iterator`. In effetti, la classe `java.util.AbstractSet` fornisce proprio queste implementazioni. Per illustrare questa tecnica, possiamo implementare il metodo `addAll` nel contesto di una classe i cui esemplari rappresentino insiemi, in questo modo:

```
public void addAll(Set<E> other) {
    for (E element : other) // usa il metodo iterator() di other
        add(element);       // i duplicati saranno ignorati da add
}
```

I metodi `retainAll` e `removeAll` possono essere realizzati con tecniche simili, anche se il metodo `retainAll` necessita di qualche attenzione in più, per evitare di eliminare elementi dallo stesso insieme di cui si sta facendo la scansione (si veda l’Esercizio C-10.59). L’efficienza di questi metodi in un’implementazione concreta del tipo di dato astratto “insieme” dipenderà dall’efficienza dei metodi fondamentali su cui si basa il loro funzionamento.

Insiemi ordinati

Nell’astrazione standard del concetto di insieme non esiste alcuna nozione esplicita del fatto che le chiavi vengano ordinate: si ipotizza solamente che il metodo `equals` possa individuare elementi equivalenti.

Se, però, gli elementi sono esemplari di una classe che implementa `Comparable` (oppure viene fornito un apposito oggetto di tipo `Comparator`), possiamo estendere il concetto di insieme per definire il tipo di dato astratto *insieme ordinato* (*sorted set*), che dichiara i seguenti metodi aggiuntivi:

- `first()`: Restituisce l’elemento minimo presente in S .
- `last()`: Restituisce l’elemento massimo presente in S .
- `ceiling(e)`: Restituisce l’elemento minimo presente in S tra quelli non minori di e .
- `floor(e)`: Restituisce l’elemento massimo presente in S tra quelli non maggiori di e .
- `lower(e)`: Restituisce l’elemento massimo presente in S tra quelli strettamente minori di e .
- `higher(e)`: Restituisce l’elemento minimo presente in S tra quelli strettamente maggiori di e .
- `subSet(e_1 , e_2)`: Restituisce un contenitore iterabile con tutti gli elementi di S non minori di e_1 e strettamente minori di e_2 .
- `pollFirst()`: Elimina da S l’elemento minimo e lo restituisce.
- `pollLast()`: Elimina da S l’elemento massimo e lo restituisce.

Nel Java Collections Framework questi metodi si trovano in una combinazione delle interfacce `java.util.SortedSet` e `java.util.NavigableSet`.

Implementazione di insiemi

Nonostante un insieme sia un'astrazione ben diversa da una mappa, le tecniche usate per implementare questi due ADT sono abbastanza simili: in effetti, un insieme è semplicemente una mappa in cui le chiavi (prive di duplicati) non sono associate ad alcun valore.

Di conseguenza, qualsiasi struttura dati usata per implementare una mappa può essere modificata per implementare il tipo di dato astratto "insieme", con la garanzia di prestazioni analoghe. Come banale adattamento di una mappa, ciascun elemento dell'insieme può essere memorizzato come chiave all'interno di una voce della mappa, usando il riferimento `null` come valore (influente). Ovviamente questa implementazione sarebbe uno spreco inutile: una realizzazione più efficiente deve abbandonare l'utilizzo degli oggetti composti di tipo `Entry` e memorizzare gli elementi dell'insieme direttamente in una struttura dati.

Nel Java Collections Framework troviamo le seguenti implementazioni di insieme, analoghe a simili strutture dati che implementano mappe:

- `java.util.HashSet` implementa l'ADT "insieme" (non ordinato) usando una tabella hash;
- `java.util.concurrent.ConcurrentSkipListSet` implementa l'ADT "insieme ordinato" usando una skip list;
- `java.util.TreeSet` implementa l'ADT "insieme ordinato" usando un albero di ricerca bilanciato (di cui parleremo nel Capitolo 11).

10.5.2 Il multi-insieme come tipo di dato astratto

Prima di parlare di modelli per l'astrazione di multi-insieme, dobbiamo prendere in esame con attenzione il concetto di elementi "duplicati". In tutto il Java Collections Framework, gli oggetti sono considerati equivalenti tra loro sulla base del normale metodo `equals` (discusso nel Paragrafo 3.5). Ad esempio, le chiavi di una mappa devono essere uniche, ma il concetto di unicità considera uguali oggetti che siano distinti ma equivalenti secondo `equals`. Questo è importante per molti utilizzi tipici delle mappe: ad esempio, quando si usano stringhe come chiavi, l'esemplare di stringa "October" che è stato usato quando si è inserita una voce nella mappa potrebbe non essere lo stesso esemplare di stringa "October" che viene usato in seguito per recuperare il valore associato a tale chiave all'interno della mappa. In tale situazione, un'invocazione come `birthstones.get("October")` avrà successo, perché le due stringhe sono considerate tra loro equivalenti.

Nel contesto dei multi-insiemi, se rappresentiamo una collezione di dati che, attraverso il concetto di equivalenza, appare descritta da $\{a, a, a, a, b, c, c\}$, dobbiamo decidere se vogliamo una struttura dati che conservi esplicitamente ciascun esemplare di a (dato che tra loro possono essere distinti, ancorché equivalenti) oppure se ci basta sapere che ne esistono quattro copie. In entrambi i casi si può implementare un multi-insieme adattando direttamente una mappa. Possiamo usare come chiave nella mappa uno degli elementi appartenenti a un gruppo di oggetti tra loro equivalenti, associando come valore un contenitore secondario con tutti gli oggetti equivalenti oppure, nella seconda ipotesi, un semplice conteggio delle sue occorrenze. Si noti che l'applicazione per il conteggio delle frequenze delle parole in un documento, vista nel Paragrafo 10.1.2, usa proprio una mappa di questo tipo, associando stringhe a conteggi.

Il Java Collections Framework non contiene alcuna forma di multi-insieme, ma esistono implementazioni di questo ADT in molte librerie Java di contenitori assai diffuse e *open source*. La libreria Apache Commons definisce le interfacce `Bag` e `SortedBag`, che corrispondono, rispettivamente, a multi-insiemi non ordinati e ordinati. Le Google Core Libraries for Java (note con il nome *Guava*) contengono, per quanto riguarda queste astrazioni, le interfacce `Multiset` e `SortedMultiset`. In entrambe queste librerie si è scelto di usare, come modello di multi-insieme, una collezione di elementi dotati di molteplicità; entrambe mettono a disposizione parecchie implementazioni concrete usando strutture dati standard. Per formalizzare il tipo di dato astratto multi-insieme, l'interfaccia `Multiset` della libreria Guava definisce (tra gli altri) i seguenti comportamenti:

- `add(e)`: Aggiunge al multi-insieme una singola occorrenza di `e`.
- `contains(e)`: Restituisce `true` se e solo se il multi-insieme contiene un elemento uguale a `e`.
- `count(e)`: Restituisce il numero di occorrenze di `e` nel multi-insieme.
- `remove(e)`: Elimina dal multi-insieme una singola occorrenza di `e`.
- `remove(e, n)`: Elimina dal multi-insieme `n` occorrenze di `e`.
- `size()`: Restituisce il numero di elementi presenti nel multi-insieme (contando anche i duplicati).
- `iterator()`: Restituisce un contenitore iterabile con tutti gli elementi del multi-insieme (ripetendo quelli che hanno molteplicità maggiore di uno).

Il tipo di dato astratto “multi-insieme” definisce anche il concetto di una “voce” (`Entry`) immutabile che rappresenta un elemento e la sua molteplicità, mentre l’interfaccia `SortedMultiset` contiene ulteriori metodi, come `firstEntry` e `lastEntry`.

10.5.3 Il tipo di dato astratto multi-mappa

Come una mappa, una multi-mappa memorizza “voci” che sono coppie chiave-valore (k, v) , dove k è la chiave e v è il valore. Mentre, però, una mappa garantisce che le sue voci abbiano chiavi univoche, una multi-mappa consente a più voci di avere la stessa chiave, in analogia con un dizionario linguistico, che ammette più definizioni di una stessa parola. In una multi-mappa, quindi, ci potranno essere due coppie, (k, v) e (k, v') , che hanno la stessa chiave k .

Ci sono due approcci standard per rappresentare una multi-mappa come variante di una mappa tradizionale. Il primo prevede di riprogettare la struttura dati interna per fare in modo che due coppie come (k, v) e (k, v') possano essere memorizzate in due voci diverse, l’altro mette in corrispondenza, in una mappa, una chiave k con un contenitore secondario i cui elementi sono tutti i valori associati a quella chiave (ad esempio, $\{v, v'\}$).

In modo molto simile alla mancanza di una definizione formale di multi-insieme, nel Java Collections Framework non troviamo neanche un’interfaccia che definisca la multi-mappa, né sono presenti implementazioni di questo tipo di dato astratto. Tuttavia, come dimostreremo a breve, è facile rappresentare una multi-mappa adattando altre classi che rappresentano contenitori e che sono presenti nel pacchetto `java.util`.

Per formalizzare la definizione del tipo di dato astratto multi-mappa, consideriamo una versione semplificata dell'interfaccia `Multimap` inclusa nella libreria Guava di Google. Tra i suoi metodi, troviamo i seguenti:

- `get(k)`: Restituisce una collezione contenente tutti i valori associati alla chiave k nella multi-mappa.
- `put(k, v)`: Aggiunge alla multi-mappa una nuova voce che associa la chiave k al valore v , senza sovrascrivere eventuali voci già esistenti con la chiave k .
- `remove(k, v)`: Elimina dalla multi-mappa una delle voci aventi chiave k e valore v (se ne esiste almeno una).
- `removeAll(k)`: Elimina dalla multi-mappa tutte le voci aventi chiave k .
- `size()`: Restituisce il numero di voci presenti nella multi-mappa (contando le associazioni multiple).
- `entries()`: Restituisce un contenitore con tutte le voci della multi-mappa.
- `keys()`: Restituisce un contenitore con le chiavi di tutte le voci della multi-mappa (con chiavi duplicate, se necessario).
- `keySet()`: Restituisce un contenitore privo di duplicati con tutte le chiavi della multi-mappa.
- `values()`: Restituisce un contenitore con tutti i valori presenti nelle voci della multi-mappa.

Nel Codice 10.16 e 10.17 presentiamo l'implementazione di una classe, `HashMultimap`, che usa un esemplare di `java.util.HashMap` per mettere in corrispondenza ciascuna chiave di una multi-mappa con un contenitore secondario, esemplare di `ArrayList`, in cui si trovano tutti i valori associati a tale chiave. Per brevità omettiamo la definizione formale dell'interfaccia `Multimap`; inoltre, definiamo il solo metodo `entries()` come unica forma di iterazione.

Codice 10.16: Un'implementazione di multi-mappa adattando classi definite nel pacchetto `java.util` (prosegue nel Codice 10.17).

```

1  public class HashMultimap<K,V> {
2      Map<K,List<V>> map = new HashMap<>(); // la mappa principale
3      int total = 0;                      // numero totale di voci nella multi-mappa
4      /** Costruisce una multi-mappa vuota. */
5      public HashMultimap() { }
6      /** Restituisce il numero totale di voci nella multi-mappa. */
7      public int size() { return total; }
8      /** Restituisce true se e solo se la multi-mappa è vuota. */
9      public boolean isEmpty() { return (total == 0); }
10     /** Restituisce un contenitore con tutti i valori associati alla chiave. */
11     public Iterable<V> get(K key) {
12         List<V> secondary = map.get(key);
13         if (secondary != null)
14             return secondary;
15         return new ArrayList<>(); // restituisce una lista di valori vuota
16     }

```

Codice 10.17: Un'implementazione di multi-mappa adattando classi definite nel pacchetto java.util (continua dal Codice 10.16).

```

17  /** Aggiunge una nuova voce che associa key a value. */
18  public void put(K key, V value) {
19      List<V> secondary = map.get(key);
20      if (secondary == null) {
21          secondary = new ArrayList<>();
22          map.put(key, secondary); // usa una nuova lista come struttura secondaria
23      }
24      secondary.add(value);
25      total++;
26  }
27  /** Elimina la voce (key,value), se esiste. */
28  public boolean remove(K key, V value) {
29      boolean wasRemoved = false;
30      List<V> secondary = map.get(key);
31      if (secondary != null) {
32          wasRemoved = secondary.remove(value);
33          if (wasRemoved) {
34              total--;
35              if (secondary.isEmpty())
36                  map.remove(key); // elimina la struttura secondaria dalla mappa principale
37          }
38      }
39      return wasRemoved;
40  }
41  /** Elimina tutte le voci aventi la chiave data. */
42  Iterable<V> removeAll(K key) {
43      List<V> secondary = map.get(key);
44      if (secondary != null) {
45          total -= secondary.size();
46          map.remove(key);
47      } else
48          secondary = new ArrayList<>(); // lista vuota di valori rimossi
49      return secondary;
50  }
51  /** Restituisce un contenitore con tutte le voci della multi-mappa. */
52  public Iterable<Map.Entry<K,V>> entries() {
53      List<Map.Entry<K,V>> result = new ArrayList<>();
54      for (Map.Entry<K,List<V> > secondary : map.entrySet()) {
55          K key = secondary.getKey();
56          for (V value : secondary.getValue())
57              result.add(new AbstractMap.SimpleEntry<K,V>(key,value));
58      }
59      return result;
60  }
61 }
```

10.6 Esercizi

Riepilogo e approfondimento

- R-10.1 Nel caso peggiore, qual è il tempo d'esecuzione richiesto per l'inserimento di n coppie chiave-valore in una mappa M inizialmente vuota, implementata con la classe `UnsortedTableMap`?

- R-10.2 Implementare di nuovo la classe `UnsortedTableMap` usando un oggetto di tipo `PositionalList` (come definito nel Paragrafo 7.3) invece di un esemplare di `ArrayList`.
- R-10.3 L'uso di valori `null` in una mappa è problematico, perché non c'è modo di distinguere tra un valore `null` restituito dall'invocazione `get(k)` che rappresenti l'effettivo valore di una voce (k, null) e un valore `null` che segnali il fatto che la chiave k non è stata trovata. L'interfaccia `java.util.Map` contiene un metodo booleano, `containsKey(k)`, che risolve completamente tale ambiguità. Implementare tale metodo nella nostra classe `UnsortedTableMap`.
- R-10.4 Tra gli schemi presentati per la gestione delle collisioni in una tabella hash, quali possono tollerare un fattore di carico maggiore di 1 e quali no?
- R-10.5 Quale sarebbe un buon codice di hash per il numero di identificazione di un veicolo, che è una stringa di numeri e lettere avente il formato "9X9XX99X9XX999999", dove un "9" rappresenta una cifra e una "X" rappresenta una lettera?
- R-10.6 Disegnare la tabella hash con 11 voci che si ottiene usando la funzione di hash $h(i) = (3i + 5) \bmod 11$ e le chiavi 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 e 5, nell'ipotesi che le collisioni siano gestite mediante concatenazione.
- R-10.7 Qual è il risultato dell'esercizio precedente se le collisioni vengono gestite mediante esplorazione lineare?
- R-10.8 Fino al punto in cui il metodo fallisce, qual è il risultato dell'esercizio R-10.6 se le collisioni vengono gestite mediante esplorazione quadratica?
- R-10.9 Qual è il risultato dell'esercizio R-10.6 se le collisioni vengono gestite mediante *double hashing* usando la funzione di hash secondaria $h'(k) = 7 - (k \bmod 7)$?
- R-10.10 Qual è il tempo richiesto nel caso peggiore per l'inserimento di n voci in una tabella hash inizialmente vuota, con collisioni risolte mediante concatenazione? E nel caso migliore?
- R-10.11 Mostrare il risultato prodotto dal *rehashing* della tabella hash della Figura 10.6 quando la si porta alla dimensione 19 usando la nuova funzione di hash $h(k) = 3k \bmod 17$.
- R-10.12 Modificare la classe `Pair` vista nel Codice 2.17 in modo che contenga una definizione naturale dei metodi `equals()` e `hashCode()`.
- R-10.13 Analizzare le righe 31, 32 e 33 del Codice 10.8, che riporta la nostra implementazione della classe `ChainHashMap`, dove, per aggiornare la variabile `n`, usiamo la differenza di dimensione del bucket secondario prima e dopo l'invocazione di `bucket.remove(k)`. Se sostituissimo quelle tre righe con le seguenti, il comportamento della classe sarebbe corretto? Spiegare.

```
V answer = bucket.remove(k);
if (answer != null) // la voce è stata eliminata
    n--; // decrementiamo la dimensione
```

- R-10.14 La nostra classe `AbstractHashMap` mantiene un fattore di carico $\lambda \leq 0.5$. Implementarla di nuovo in modo che l'utilizzatore possa specificare il fattore di carico massimo e modificare in modo conseguente le sue sottoclassi concrete.
- R-10.15 Fornire, mediante pseudocodice, una descrizione della procedura di inserimento in una tabella hash che risolve le collisioni mediante esplorazione quadratica, ipotizzando di usare anche il "trucco" di sostituire le voci rimosse con un oggetto speciale che contrassegni le posizioni "disponibili".

- R-10.16 Modificare la nostra classe `ProbeHashMap` in modo che usi l'esplorazione quadratica.
- R-10.17 Spiegare perché una tabella hash non è adatta all'implementazione di una mappa ordinata.
- R-10.18 Nel caso peggiore, qual è il tempo d'esecuzione asintotico per effettuare n rimozioni da un esemplare di `SortedTableMap` che contenga inizialmente $2n$ voci?
- R-10.19 Implementare nella classe `SortedTableMap` il metodo `containsKey(k)` descritto nell'Esercizio R-10.3.
- R-10.20 Descrivere come una lista ordinata realizzata mediante lista doppiamente concatenata possa essere utilizzata per implementare il tipo di dato astratto "mappa ordinata".
- R-10.21 Considerare la seguente variante del metodo `findIndex` della classe `SortedTableMap`, originariamente descritta nel Codice 10.11:

```

1  private int findIndex(K key, int low, int high) {
2      if (high < low) return high + 1;
3      int mid = (low + high) / 2;
4      if (compare(key, table.get(mid)) < 0)
5          return findIndex(key, low, mid - 1);
6      else
7          return findIndex(key, mid + 1, high);
8  }

```

Questo metodo produce sempre lo stesso risultato prodotto dalla versione originale? Giustificare la risposta.

- R-10.22 Qual è il tempo d'esecuzione atteso dei metodi che gestiscono un insieme di massimi se inseriamo n coppie tali che ciascuna di esse abbia costo e prestazione inferiori rispetto a una delle coppie inserite in precedenza? Cosa contiene la mappa ordinata alla fine di questa sequenza di operazioni? Cosa succederebbe se, invece, ciascuna coppia avesse un costo inferiore e una prestazione maggiore rispetto a una delle coppie inserite in precedenza?
- R-10.23 Disegnare il risultato prodotto dall'esecuzione della seguente sequenza di operazioni sulla skip list della Figura 10.13: `remove(38)`, `put(48, x)`, `put(24, y)`, `remove(55)`. Usare un vero lancio di moneta per generare le scelte casuali necessarie (e indicare nel disegno la sequenza di risultati ottenuti con i lanci).
- R-10.24 Fornire, mediante pseudocodice, una descrizione dell'operazione `remove` per una mappa realizzata con una skip list.
- R-10.25 Fornire, mediante pseudocodice, una descrizione dell'implementazione del metodo `removeAll` per il tipo di dato astratto "insieme", usando soltanto gli altri metodi fondamentali dell'insieme.
- R-10.26 Fornire, mediante pseudocodice, una descrizione dell'implementazione del metodo `retainAll` per il tipo di dato astratto "insieme", usando soltanto gli altri metodi fondamentali dell'insieme.
- R-10.27 Se indichiamo con n la dimensione dell'insieme S e con m la dimensione dell'insieme T , qual è il tempo d'esecuzione dell'operazione `S.addAll(T)`, descritta nel Paragrafo 10.5.1, se entrambi gli insiemi sono implementati con una skip list?

- R-10.28 Se indichiamo con n la dimensione dell'insieme S e con m la dimensione dell'insieme T , qual è il tempo d'esecuzione dell'operazione $S.addAll(T)$, descritta nel Paragrafo 10.5.1, se entrambi gli insiemi sono implementati usando una strategia di hashing?
- R-10.29 Se indichiamo con n la dimensione dell'insieme S e con m la dimensione dell'insieme T , qual è il tempo d'esecuzione dell'operazione $S.removeAll(T)$ se entrambi gli insiemi sono implementati usando una strategia di hashing?
- R-10.30 Se indichiamo con n la dimensione dell'insieme S e con m la dimensione dell'insieme T , qual è il tempo d'esecuzione dell'operazione $S,retainAll(T)$ se entrambi gli insiemi sono implementati usando una strategia di hashing?
- R-10.31 Quale astrazione usereste per gestire una base di dati relativa ai compleanni dei vostri amici in modo che si possano effettuare in modo efficiente interrogazioni come "trova tutti gli amici che compiono gli anni oggi" e "trova chi sarà il prossimo a compiere gli anni"?

Creatività

- C-10.32 Per ottenere una funzione di compressione ideale, la capacità dell'array di bucket di una tabella hash dovrebbe essere un numero primo, quindi consideriamo il problema di individuare un numero primo appartenente all'intervallo $[M, 2M]$. Implementare un metodo che trovi un tale numero primo usando l'*algoritmo del setaccio* (*sieve algorithm*): creiamo un array A di $2M$ celle booleane, in modo che la cella i sia associata al numero intero i , poi inizializziamo le celle dell'array in modo che contengano tutte `true`, quindi contrassegniamo con `false` le celle che corrispondono ai multipli di 2, ai multipli di 3, ai multipli di 5, ai multipli di 7 e così via. La procedura termina dopo aver raggiunto un numero maggiore di $(2M)^{1/2}$ (*suggerimento*: considerare un metodo ausiliario che trovi i numeri primi fino a $(2M)^{1/2}$).
- C-10.33 Consideriamo l'obiettivo di aggiungere la voce (k, v) a una mappa soltanto se non esiste già una voce con chiave k . In una mappa M (che non ammetta valori `null`), questo obiettivo si potrebbe raggiungere in questo modo:

```
if (M.get(k) == null)
    M.put(k, v);
```

Anche se questo codice raggiunge l'obiettivo, la sua efficienza non è ideale, perché il tempo dedicato a una ricerca fallita durante l'invocazione di `get` verrà speso di nuovo durante l'invocazione di `put` (che, di nuovo, inizierà cercando di trovare la posizione di una voce avente la chiave data). Per evitare questa inefficienza, alcune implementazioni di mappa mettono a disposizione un metodo apposito, `putIfAbsent(k, v)`, che evita questo spreco di tempo. Implementare tale metodo nella classe `UnsortedTableMap`.

- C-10.34 Ripetere l'Esercizio C-10.33 per la classe `ChainHashMap`.
- C-10.35 Ripetere l'Esercizio C-10.33 per la classe `ProbeHashMap`.
- C-10.36 Descrivere come riprogettare la classe di base `AbstractHashMap` perché contenga anche il metodo `containsKey` descritto nell'Esercizio R-10.3.

- C-10.37 Modificare la classe `ChainHashMap` secondo quanto progettato nell'esercizio precedente.
- C-10.38 Modificare la classe `ProbeHashMap` secondo quanto progettato nell'Esercizio C-10.36.
- C-10.39 Riprogettare la classe di base `AbstractHashMap` in modo che dimezzi la capacità della tabella se il fattore di carico scende al di sotto di 0.25. La soluzione proposta non deve rendere necessaria alcuna modifica alle classi concrete `ProbeHashMap` e `ChainHashMap`.
- C-10.40 La classe `java.util.HashMap` usa la concatenazione separata senza alcuna struttura secondaria esplicita: la tabella è un array di voci (*entry*), ciascuna delle quali dispone di un campo aggiuntivo, `next`, che può fare riferimento a un'altra voce di quel bucket. In questo modo gli esemplari di voci possono dar vita a una lista semplicemente concatenata. Implementare nuovamente la nostra classe `ChainHashMap` in modo che usi questo approccio.
- C-10.41 Descrivere come si esegue una rimozione da una tabella hash che usa l'esplorazione lineare per risolvere le collisioni, senza però utilizzare un oggetto speciale per rappresentare gli elementi rimossi. Bisogna, cioè, spostare il contenuto della mappa in modo che tutto vada come se l'elemento rimosso non fosse mai stato presente nella posizione in cui era.
- C-10.42 La strategia di esplorazione quadratica soffre di un problema di *clustering* relativo al modo in cui cerca le posizioni disponibili. Nello specifico, quando avviene una collisione nel bucket $h(k)$, si verificano i bucket $A[(h(k) + i^2) \bmod N]$, con $i = 1, 2, \dots, N - 1$.
- Dimostrare che l'espressione $(i^2 \bmod N)$ assume al massimo $(N + 1)/2$ valori diversi, se N è un numero primo, al variare di i da 1 a $N - 1$. Come parte di questa dimostrazione, osservare che $i^2 \bmod N = (N - i)^2 \bmod N$ per qualsiasi valore di i .
 - Una strategia migliore prevede di scegliere un numero primo N tale che $N \bmod 4 = 3$ e, poi, verificare la disponibilità dei bucket $A[(h(k) \pm i^2) \bmod N]$, con i che varia da 1 a $(N - 1)/2$, scegliendo alternativamente il segno più e il segno meno. Dimostrare che questa versione alternativa garantisce l'esplorazione di tutti i bucket presenti in A .
- C-10.43 Progettare nuovamente la classe `ProbeHashMap` in modo che sia più semplice personalizzare la sequenza di esplorazione dei bucket durante la risoluzione di una collisione. Illustrare l'efficacia del nuovo progetto rendendo disponibili sottoclassi concrete distinte per l'esplorazione lineare e quadratica.
- C-10.44 La classe `java.util.LinkedHashMap` è una sottoclasse della classe `HashMap` che preserva le prestazioni attese $O(1)$ per le principali operazioni della mappa, garantendo che le iterazioni scandiscano le voci della mappa secondo una politica FIFO (*first-in first-out*), nel senso che la prima chiave a essere restituita dalle iterazioni è quella che si trova all'interno della mappa da più tempo (l'ordine temporale tra le chiavi non è influenzato da eventuali modifiche avvenute ai valori ad esse associati). Descrivere un approccio logaritmico per raggiungere questo obiettivo.

- C-10.45 Sviluppare una versione “consapevole della posizione” (*location-aware*) della classe `UnsortedTableMap`, in modo che l’operazione `remove(e)` per una voce e presente nella mappa possa essere eseguita in un tempo $O(1)$.
- C-10.46 Ripetere l’esercizio precedente per la classe `ProbeHashMap`.
- C-10.47 Ripetere l’Esercizio C-10.45 per la classe `ChainHashMap`.
- C-10.48 Anche se in una mappa le chiavi sono distinte, l’algoritmo di ricerca binaria può essere applicato in un contesto più generale, nel quale un array memorizza elementi in ordine non decrescente, con possibili duplicati. Consideriamo l’obiettivo di individuare l’indice dell’elemento *più a sinistra* avente chiave non minore di un valore assegnato, k . Il metodo `findIndex` definito nel Codice 10.11 garantisce di trovare la soluzione? Lo fa il metodo `findIndex` descritto nell’Esercizio R-10.21? Giustificare le risposte fornite.
- C-10.49 Date due tabelle di ricerca ordinate, S e T , ciascuna con n voci e implementate mediante array, descrivere un algoritmo che in un tempo $O(\log^2 n)$ trovi la k -esima chiave più piccola nell’insieme unione delle chiavi di S e di T (nell’ipotesi che in tale unione non siano presenti chiavi duplicate).
- C-10.50 Fornire una soluzione $O(\log n)$ al problema precedente.
- C-10.51 Progettare un’implementazione alternativa per il metodo `entrySet` della classe `SortedTableMap` che crei un *iteratore pigro* (*lazy iterator*) invece che un iteratore che opera su una fotografia istantanea (*snapshot*), come spiegato nel Paragrafo 7.4.2 dedicato alle due forme di iteratori.
- C-10.52 Ripetere l’esercizio precedente per la classe `ChainHashMap`.
- C-10.53 Ripetere l’Esercizio C-10.51 per la classe `ProbeHashMap`.
- C-10.54 Data una base di dati D contenente n coppie costo-prestazione (c, p) , descrivere un algoritmo che trovi le coppie massime di D in un tempo $O(n \log n)$.
- C-10.55 Mettere in evidenza che i metodi `above(p)` e `before(p)` non sono per nulla necessari per implementare in modo efficiente una mappa usando una skip list, perché è possibile implementare le operazioni di inserimento e rimozione in una skip list con un approccio che proceda rigidamente dall’alto in basso e da sinistra a destra, senza dover mai usare i metodi `above` e `before` (*suggerimento*: nell’algoritmo di inserimento, per prima cosa lanciare ripetutamente una moneta per determinare il livello in cui iniziare l’inserimento della nuova voce).
- C-10.56 Descrivere come si possa modificare la struttura dati che memorizza una skip list per fornire adeguato supporto al metodo `median()`, che restituisce la posizione dell’elemento che, nella lista “più bassa” S_0 , corrisponde all’indice $\lfloor n/2 \rfloor$. Dimostrare che l’implementazione proposta viene eseguita in un tempo atteso $O(\log n)$.
- C-10.57 Descrivere come si possa modificare la rappresentazione di una skip list in modo che le operazioni basate su indici, come l’ispezione della voce corrispondente all’indice j , possano essere eseguite in un tempo atteso $O(\log n)$.
- C-10.58 Ogni riga di un array A avente n righe e n colonne (un array $n \times n$) è costituita da cifre 1 e 0 in modo che tutte le cifre 1 precedano tutte le cifre 0. Nell’ipotesi che A si trovi già in memoria, descrivere un metodo che conti il numero di cifre 1 presenti in A in un tempo $O(n \log n)$ (e non $O(n^2)$).
- C-10.59 Fornire un’implementazione concreta del metodo `retainAll` dell’ADT “insieme” usando soltanto gli altri metodi fondamentali dell’insieme, nell’ipotesi che l’implementazione di insieme usi *iteratori di tipo fail-fast*, descritti nel Paragrafo 7.4.2.

- C-10.60 Prendiamo in esame insiemi i cui elementi siano numeri interi appartenenti all'intervallo $[0, N - 1]$. Uno schema molto comune per rappresentare un insieme A di questo tipo prevede l'utilizzo di un array booleano, B , seguendo la convenzione che x appartiene all'insieme A se e solo se $B[x] = \text{true}$. Dato che ogni cella di B può essere rappresentata da un singolo bit, a volte si parla di rappresentazione mediante *vettore di bit*. Descrivere e analizzare algoritmi efficienti per eseguire i metodi dell'ADT "insieme" usando tale rappresentazione.
- C-10.61 Un *file invertito* (*inverted file*) è una struttura dati cruciale per l'implementazione di applicazioni come un motore di ricerca o l'indice analitico di un libro. Dato un documento D , che può essere visto come una lista numerata e non ordinata di parole, un file invertito è una lista ordinata di parole, L , tale che, per ogni parola w in L , si memorizzano gli indici delle posizioni in D in cui compare w . Progettare un algoritmo efficiente per costruire L a partire da D .
- C-10.62 L'operazione $\text{get}(k)$ per il tipo di dato astratto multi-mappa ha il compito di restituire un contenitore con *tutti* i valori associati alla chiave k . Progettare una variante della ricerca binaria che esegua questa operazione in una tabella di ricerca ordinata che può contenere duplicati e dimostrare che il suo tempo d'esecuzione è $O(s + \log n)$, dove n è il numero di elementi nella multi-mappa e s è il numero di voci aventi chiave k .
- C-10.63 Descrivere una struttura efficiente per implementare una multi-mappa avente n voci associate complessivamente a $r < n$ chiavi appartenenti a un insieme totalmente ordinato: in pratica, l'insieme delle chiavi ha un numero di elementi minore del numero di voci presenti nella multi-mappa. La struttura deve eseguire: l'operazione get in un tempo atteso $O(s + \log r)$, se s è il numero di voci restituite; l'operazione entries in un tempo $O(n)$; le restanti operazioni della multi-mappa in un tempo atteso $O(\log r)$.
- C-10.64 Descrivere una struttura efficiente per implementare una multi-mappa avente n voci associate complessivamente a $r < n$ chiavi che abbiano codici di hash distinti. La struttura deve eseguire: l'operazione get in un tempo atteso $O(s + 1)$, se s è il numero di voci restituite; l'operazione entries in un tempo $O(n)$; le restanti operazioni della multi-mappa in un tempo atteso $O(1)$.

Progettazione

- P-10.65 Una strategia interessante per gestire una tabella hash mediante indirizzamento aperto è nota come *cuckoo hashing* (*hashing del cicalo*, per l'abitudine che tale volatile ha di "sfrattare" altri uccelli dal nido per occuparlo). Per ogni chiave vengono calcolate due funzioni di hash indipendenti e una voce viene sempre memorizzata in una delle due celle indicate da tali funzioni applicate alla propria chiave. Quando viene inserita una nuova voce, se una delle due celle indicate è disponibile, viene memorizzata là, altrimenti viene posta a caso in una delle due posizioni selezionate dalle funzioni di hash, sfrattando la voce ivi presente. La voce sfrattata viene poi posizionata nell'altra cella che le compete, eventualmente sfrattando un'altra voce, e via così, continuando finché non si riesce a posizionare una voce in una cella vuota oppure si individua un ciclo infinito (nel qual caso vengono scelte due nuove funzioni di hash e tutte le voci presenti nella tabella vengono eliminate e reinserite). Si può dimostrare che, almeno finché il fattore di carico della tabella rimane al di sotto di 0.5, un inserimento ha successo in un tempo

atteso costante. Si osservi che si può eseguire una ricerca in un tempo costante nel caso peggiore, perché l'elemento cercato può essere memorizzato soltanto in una di due sole posizioni possibili. Progettare un'implementazione completa di una mappa basata su questa strategia.

- P-10.66 Una strategia interessante per gestire una tabella hash con concatenazione separata prende il nome di *power-of-two-choices hashing*. Per ogni chiave vengono calcolate due funzioni di hash indipendenti e una nuova voce da inserire viene memorizzata nel bucket, tra i due indicati dalla funzioni di hash, che ha il minor numero di voci. Progettare un'implementazione completa di una mappa basata su questa strategia.
- P-10.67 Implementare la classe `LinkedHashMap`, come descritto nell'Esercizio C-10.44, garantendo che le operazioni sulla mappa principale vengano eseguite in un tempo atteso $O(1)$.
- P-10.68 Eseguire esperimenti sulle nostre classi `ChainHashMap` e `ProbeHashMap` per misurarne l'efficienza, usando insiemi di chiavi casuali e variando la soglia del fattore di carico (si veda l'Esercizio R-10.14).
- P-10.69 Eseguire un'analisi comparativa che studi le frequenze di collisione per varie codifiche di hash progettate per stringhe di caratteri, come le codifiche polinomiali con diversi valori del parametro a . Usare una tabella hash per determinare il numero di collisioni, ma contare soltanto le collisioni dovute al fatto che a stringhe diverse venga attribuito lo stesso codice di hash (e non quando stringhe con codici di hash diversi vengono assegnate alla stessa posizione nella tabella). Verificare queste codifiche di hash con file di testo trovati in Internet.
- P-10.70 Eseguire un'analisi comparativa come quella descritta nell'esercizio precedente, usando numeri di telefono a 10 cifre invece di stringhe di caratteri.
- P-10.71 Progettare una classe, in Java, che implementi la struttura dati *skip list*. Usare, poi, tale classe per definire un'implementazione completa dell'ADT "mappa ordinata".
- P-10.72 Estendere il progetto precedente aggiungendo un'animazione grafica delle operazioni della skip list. Visualizzare il modo in cui le voci risalgono la skip list durante un inserimento e vengono, invece, estromesse durante una rimozione. Inoltre, nelle operazioni di ricerca, visualizzare le azioni di scansione in avanti e di discesa.
- P-10.73 Descrivere come si possa usare una skip list per implementare il tipo di dato astratto "lista con indice" (*array list*), in modo che le operazioni di inserimento e rimozione basate su indici vengano entrambe eseguite in un tempo atteso $O(\log n)$.
- P-10.74 Scrivere una classe che svolga il ruolo di verificatore lessicale (*spell checker*) memorizzando un dizionario di parole, W , all'interno di un insieme e implementando un metodo, `check(s)`, che esegua la verifica lessicale della stringa s rispetto all'insieme di parole W . Se s appartiene a W , l'invocazione `check(s)` restituisce una lista contenente soltanto s , perché in tal caso si ipotizza che sia stata scritta correttamente. Se, invece, s non è presente in W , allora l'invocazione `check(s)` restituisce una lista contenente tutte le parole di W che potrebbero essere una forma corretta di s . Il programma deve essere in grado di gestire tutti gli errori più frequenti che possono accadere quando si sbaglia una parola, ad esempio lo scambio di caratteri adiacenti, l'inserimento di un unico carattere estraneo fra due caratteri adiacenti, la mancanza di un carattere e la sostituzione di un carattere con un altro carattere. Come ulteriore elemento di difficoltà, considerare anche le sostituzioni fonetiche.

Note

La tecnica di hashing è stata molto studiata. Al lettore interessato ad approfondirne lo studio suggeriamo il libro di Knuth [61] e quello di Vitter e Chen [92]. La vulnerabilità di sicurezza di tipo *denial-of-service* che sfrutta le prestazioni nel caso peggiore delle tabelle hash è stata descritta per la prima volta da Crosby e Wallach [27], poi illustrata in dettaglio da Klink e Wälde [58]; la contromisura adottata per Java dal gruppo di sviluppo OpenJDK è descritta in [76].

La skip list è stata presentata da Pugh [80]. La nostra analisi della skip list è una forma semplificata di quella fornita da Motwani e Raghavan [75]. Per un'analisi più approfondita, si possono consultare i molti lavori scientifici sulle skip list che sono apparsi in letteratura [56, 77, 78]. L'Esercizio C-10.42 è stato elaborato da James Lee.

11

Alberi di ricerca

11.1 Alberi di ricerca binari

Nel Capitolo 8 abbiamo introdotto la struttura dati ad albero, illustrandone varie applicazioni. Un suo utilizzo molto importante è l'*albero di ricerca (search tree)*, già descritto nel Paragrafo 8.4.3, e in questo capitolo useremo una struttura ad albero di ricerca per implementare in modo efficiente una *mappa ordinata*. I tre metodi fondamentali di una mappa (visti nel Paragrafo 10.1.1) sono:

- get(k):** Restituisce il valore v associato alla chiave k , se esiste la voce (k, v) , altrimenti restituisce **null**.
- put(k, v):** Associa il valore v alla chiave k , sostituendo e restituendo un eventuale valore precedentemente associato a k .
- remove(k):** Se esiste, elimina la voce avente chiave uguale a k e ne restituisce il valore, altrimenti restituisce **null**.

Il tipo di dato astratto “mappa ordinata” definisce funzionalità ulteriori, come visto nel Paragrafo 10.3, garantendo che le iterazioni agenti sulla mappa restituiscano le chiavi in ordine e fornendo anche supporto per ulteriori ricerche, come `higherEntry(k)` o `subMap(k_1, k_2)`.

Gli alberi binari sono una struttura dati eccellente per memorizzare le voci di una mappa, nell’ipotesi che tra le sue chiavi sia definita una relazione d’ordine. In questo capitolo definiamo un *albero di ricerca binario (binary search tree)* come un *albero binario proprio* (si veda il Paragrafo 8.2) tale che ogni posizione interna p memorizzi una coppia chiave-valore (k, v) tale che:

- Le chiavi memorizzate nel sottoalbero sinistro di p sono minori di k .
- Le chiavi memorizzate nel sottoalbero destro di p sono maggiori di k .

La Figura 11.1 riporta un esempio di albero di ricerca binario. Si noti come le foglie dell'albero servano soltanto come "segnaposto": il loro uso come sentinelle semplifica la definizione di alcuni dei nostri algoritmi di ricerca e modifica. In pratica, facendo un po' di attenzione, possono essere rappresentate come riferimenti `null`, riducendo così il numero di nodi alla metà (perché in un albero binario proprio ci sono più foglie che nodi interni).

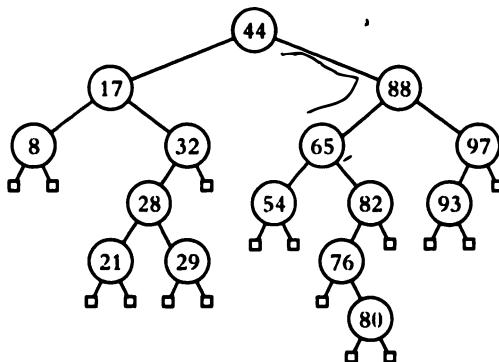


Figura 11.1: Un albero di ricerca binario con numeri interi come chiavi. In questo intero capitolo eviteremo di visualizzare i valori associati alle chiavi, perché non influenzano la disposizione delle voci all'interno di un albero di ricerca.

11.1.1 Ricerca in un albero di ricerca binario

La conseguenza più importante della proprietà strutturale dell'albero di ricerca binario è l'omonimo algoritmo di ricerca. Possiamo cercare di individuare la posizione di una particolare chiave all'interno di un albero di ricerca binario trattandolo come se fosse un albero di decisione (descritto nella Figura 8.5). In questo caso, la domanda che viene posta in corrispondenza di ciascuna posizione interna p riguarda il fatto che la chiave cercata, k , sia minore della chiave memorizzata nella posizione p (che indichiamo con $\text{key}(p)$), oppure sia uguale o, ancora, sia maggiore. Se la risposta è "minore di", allora la ricerca continua nel sottoalbero sinistro. Se la risposta è "uguale a", la ricerca termina con successo. Se, infine, la risposta è "maggiore di", allora la ricerca continua nel sottoalbero destro. Se, così facendo, arriviamo in una foglia, la ricerca termina senza successo, come evidenziato nella Figura 11.2.

Nel Codice 11.1 descriviamo proprio questo approccio. Se la chiave k si trova nel sottoalbero avente radice p , l'invocazione `TreeSearch(p , k)` restituisce la posizione in cui si trova la chiave. In una ricerca infruttuosa, l'algoritmo `TreeSearch` restituisce la foglia che conclude il percorso di ricerca (e più avanti la useremo per determinare la posizione in cui inserire una nuova voce in un albero di ricerca).

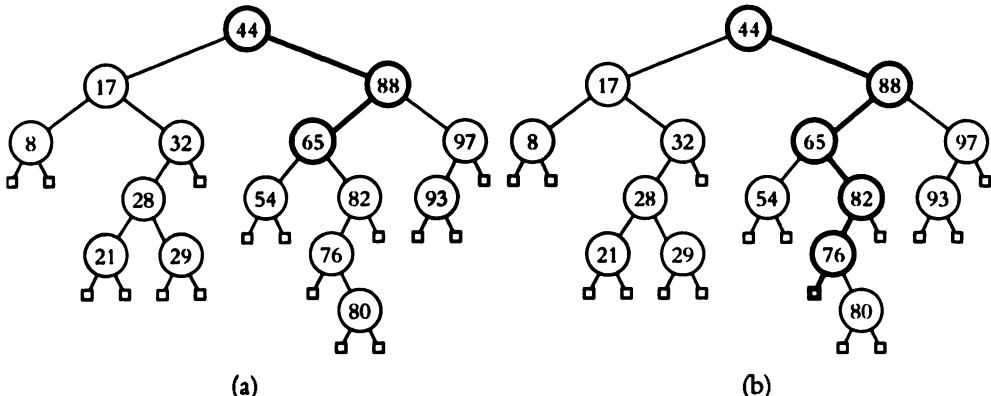


Figura 11.2: (a) Ricerca fruttuosa della chiave 65 in un albero di ricerca binario; (b) ricerca infruttuosa della chiave 68, che termina nella foglia che si trova alla sinistra della chiave 76.

Codice 11.1: Ricerca ricorsiva in un albero di ricerca binario.

Algoritmo TreeSearch(p, k):

```

if  $p$  è esterna then
    return  $p$                                 { ricerca senza successo }

else if  $k == \text{key}(p)$  then
    return  $p$                                 { ricerca con successo }

else if  $k < \text{key}(p)$  then
    return TreeSearch(left( $p$ ),  $k$ )          { ricorsione nel sottoalbero sinistro }

else { ora sappiamo che  $k > \text{key}(p)$  }
    return TreeSearch(right( $p$ ),  $k$ )          { ricorsione nel sottoalbero destro }
```

Analisi della ricerca in un albero di ricerca binario

L'analisi del tempo d'esecuzione nel caso peggiore della ricerca in un albero di ricerca binario T è semplice. L'algoritmo TreeSearch è ricorsivo ed esegue un numero costante di operazioni primitive per ogni invocazione ricorsiva. Ciascuna invocazione ricorsiva di TreeSearch è relativa al figlio della posizione precedente, quindi TreeSearch viene invocato sulle posizioni appartenenti a un percorso di T che parte dalla radice e scende ogni volta di un livello. Di conseguenza, il numero di tali posizioni esaminate è limitato superiormente da $h + 1$, dove h è l'altezza di T . In altre parole, dato che l'algoritmo impiega un tempo $O(1)$ per ogni posizione esaminata durante la ricerca, il tempo d'esecuzione complessivo è $O(h)$, dove h è l'altezza dell'albero di ricerca binario T (come si può vedere nella Figura 11.3).

In riferimento all'ADT mappa ordinata, la ricerca sarà usata come procedura ausiliaria per implementare i metodi `get`, `put` e `remove`, perché ciascuno di questi inizia cercando di individuare una voce nella mappa che abbia la chiave data. Più tardi faremo vedere come implementare le operazioni più specifiche della mappa ordinata, come `lowerEntry` e `higherEntry`, navigando nell'albero dopo l'esecuzione di una normale ricerca. Tutte queste operazioni verranno eseguite nel caso peggiore in un tempo $O(h)$ in un albero di altezza h .

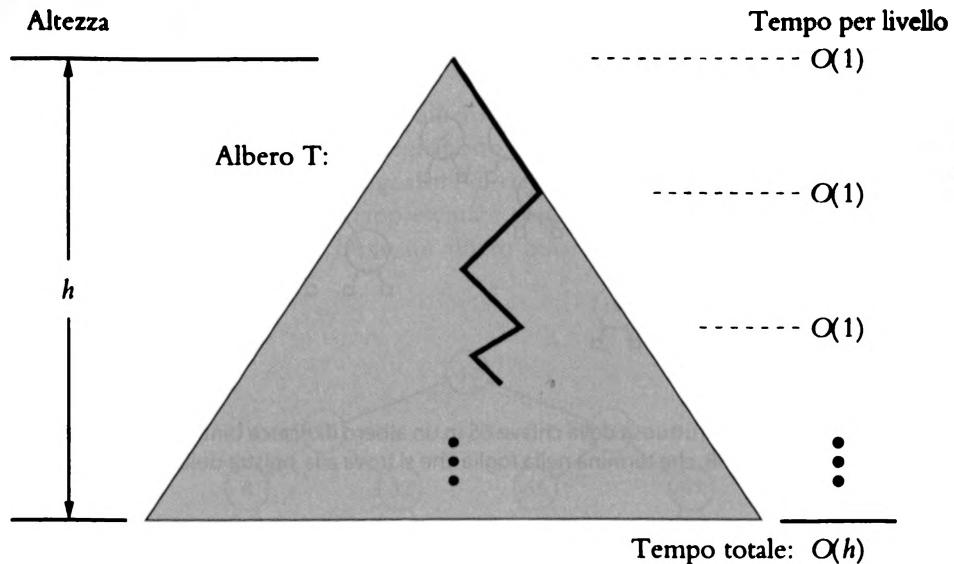


Figura 11.3: Visualizzazione del tempo d'esecuzione di una ricerca in un albero di ricerca binario. La figura usa una rappresentazione sintetica dell'albero di ricerca binario molto diffusa: un grande triangolo, con un percorso che procede a zig zag scendendo dalla radice.

Va detto che l'altezza h di T può raggiungere il numero di voci, n : il valore massimo di h è, infatti, n , ma ci aspettiamo che solitamente sia molto inferiore. Più avanti, in questo stesso capitolo, vedremo diverse strategie che consentono di limitare superiormente l'altezza di un albero di ricerca a un valore $O(\log n)$.

11.1.2 Inserimenti e rimozioni

Gli alberi di ricerca binari consentono implementazioni delle operazioni `put` e `remove` usando algoritmi che sono abbastanza semplici, anche se non proprio banali.

Inserimento

L'operazione `put(k, v)` della mappa inizia con la ricerca di una voce avente chiave k . Se la trova, a tale voce viene assegnato il nuovo valore v , altrimenti si può inserire la nuova voce nell'albero trasformando in nodo interno la foglia in cui era terminata la ricerca. La proprietà dell'albero di ricerca binario è rispettata anche dopo questo posizionamento: si noti che la nuova voce viene inserita proprio nella posizione dove l'algoritmo di ricerca si aspettava di trovarla. Ipotizziamo che un albero binario proprio metta a disposizione anche la seguente operazione di aggiornamento dell'albero:

`expandExternal(p, e)`: Memorizza la voce e nella posizione esterna p (una foglia) e trasforma p in un nodo interno, con due nuove foglie come figli.

Quindi, possiamo descrivere l'algoritmo di inserimento `TreeInsert` con lo pseudocodice presentato nel Codice 11.2. La Figura 11.4 mostra un esempio di inserimento in un albero di ricerca binario.

Codice 11.2: Algoritmo che inserisce una coppia chiave-valore in una mappa rappresentata mediante un albero di ricerca binario.

Algoritmo `TreeInsert`(k, v):

Input: Una chiave k da associare al valore v

$p = \text{TreeSearch}(\text{root}(), k)$

if $k == \text{key}(p)$ *then*

Assegna v come valore della voce memorizzata in p

else

`expandExternal`($p, (k, v)$)

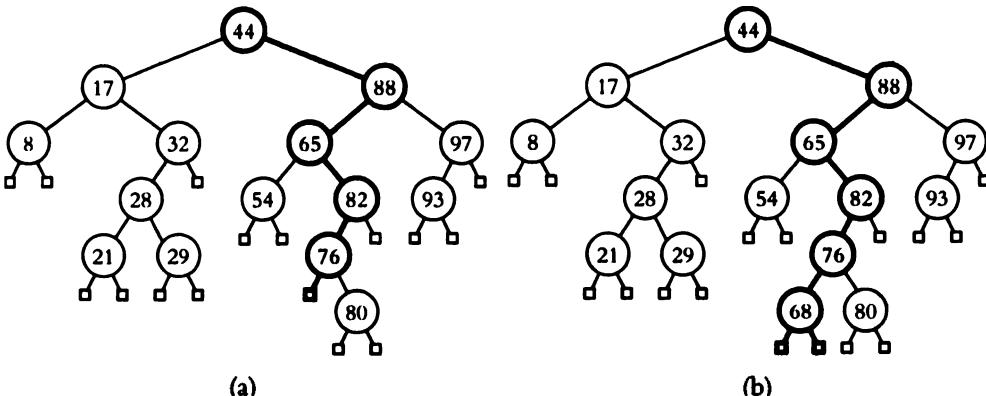


Figura 11.4: (a) Inserimento di una nuova voce con chiave 68 nell'albero di ricerca binario della Figura 11.1: la ricerca della posizione in cui inserirla è evidenziata in (a), mentre (b) mostra l'albero risultante.

Rimozione

La rimozione di una voce da un albero di ricerca binario è un po' più complessa dell'inserimento di una nuova voce, perché la posizione della voce da rimuovere può trovarsi in qualsiasi punto dell'albero (diversamente dalla posizione di inserimento, che è sempre una foglia). Per eliminare la voce contenente la chiave k , si parte con `TreeSearch`($\text{root}(), k$) per trovare la posizione p che contiene la voce cercata (se esiste). Se la ricerca restituisce una foglia, significa che non c'è alcuna voce da eliminare, altrimenti dobbiamo distinguere due casi (di difficoltà crescente):

- Se al più uno dei figli della posizione p è un nodo interno, la rimozione della voce in posizione p si realizza facilmente, come si può vedere nella Figura 11.5. Indichiamo con r il figlio interno di p (o uno qualsiasi dei due figli, se entrambi sono foglie). Elimineremo p e il fratello di r (che è una foglia), facendo risalire r perché prenda il posto di p . Osserviamo che tutte le altre relazioni di parentela tra antenati e discendenti

rimaste nell'albero dopo l'operazione esistevano anche prima dell'operazione, quindi la proprietà fondamentale dell'albero di ricerca binario è preservata.

- Se la posizione p ha due figli che sono nodi interni, non possiamo semplicemente eliminare il nodo p dall'albero, perché questo creerebbe un "buco" e lascerebbe due figli orfani. Come illustrato nella Figura 11.6, procediamo, invece, in questo modo:
 - Individuiamo la posizione r che contiene la voce avente la chiave massima tra quelle strettamente minori della chiave presente nella posizione p (il cosiddetto *predecessore* nell'ordine stabilito tra le chiavi). Tale predecessore sarà sempre posizionato nella posizione interna più a destra del sottoalbero sinistro di p .
 - Usiamo la voce memorizzata in r per sostituire la voce presente in p , che deve essere eliminata. Dato che la chiave di r precede immediatamente quella di p nell'insieme delle chiavi della mappa, ogni voce del sottoalbero destro di p avrà chiave maggiore della chiave di r e ogni altra voce del sottoalbero sinistro di p avrà chiave minore della chiave di r . Quindi, la proprietà fondamentale dell'albero di ricerca binario è preservata dopo questa sostituzione.
 - Avendo usato la voce di r come sostituta della voce di p , invece di eliminare il nodo p dall'albero possiamo eliminare il nodo r . Fortunatamente, essendo r la posizione interna più a destra in un sottoalbero, non ha un figlio destro che sia interno, quindi la sua rimozione può essere effettuata usando il primo (e più semplice) approccio.

Come con la ricerca e l'inserimento, questo algoritmo di rimozione effettua l'attraversamento di un unico percorso in discesa a partire dalla radice, eventualmente spostando un'unica voce da una posizione a un'altra appartenenti a tale percorso, per poi eliminare un nodo del percorso e facendone risalire il figlio. Quindi, l'intera procedura viene eseguita in un tempo $O(h)$, dove h è l'altezza dell'albero.

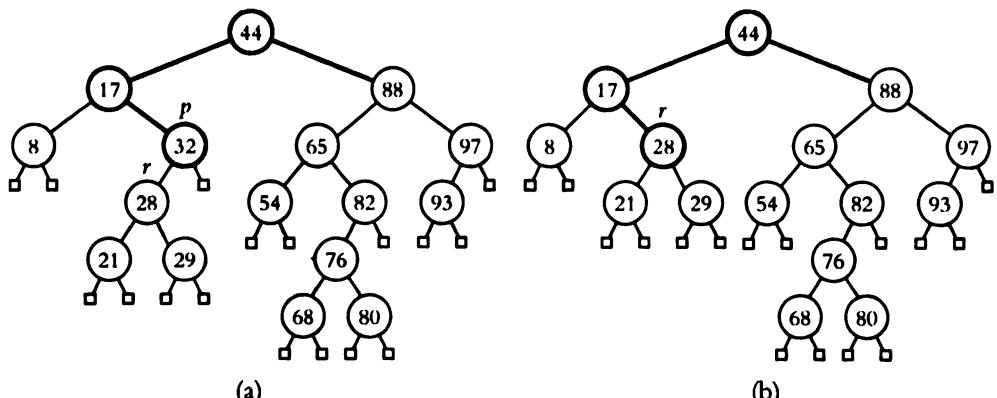


Figura 11.5: Eliminazione della voce con chiave 32 dall'albero di ricerca binario della Figura 11.4b. La voce da eliminare si trova nella posizione p , che ha un solo figlio interno, r :
 (a) prima della rimozione; (b) dopo la rimozione.

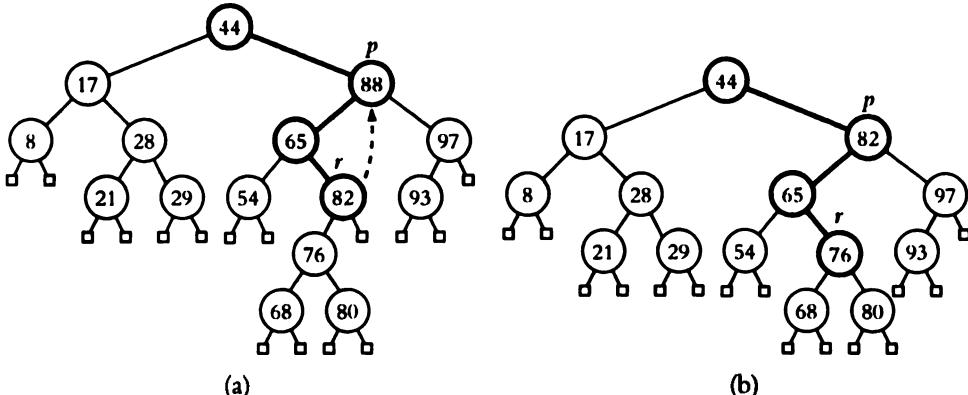


Figura 11.6: Eliminazione della voce con chiave 88 dall'albero di ricerca binario della Figura 11.5b. La voce da eliminare si trova nella posizione p , che ha due figli interni, e viene sostituita dal suo predecessore, che si trova nella posizione r . (a) prima della rimozione; (b) dopo la rimozione.

11.1.3 Implementazione in Java

Nel Codice 11.3, 11.4, 11.5 e 11.6 definiamo la classe `TreeMap` che implementa il tipo di dato astratto “mappa ordinata” usando un albero di ricerca binario per la memorizzazione delle sue voci. La classe `TreeMap` è stata dichiarata come sottoclasse della classe di base `AbstractSortedMap`, ereditando così anche il supporto per l'esecuzione di confronti (basati su un oggetto di tipo `Comparator` che venga fornito come parametro di costruzione o un analogo oggetto predefinito), la classe annidata `MapEntry` per memorizzare le coppie chiave-valore e implementazioni concrete dei metodi `keySet` e `values`, che sfruttano il metodo `entrySet` che definiremo (la Figura 10.2 presenta una panoramica dell'intera gerarchia di ereditarietà delle mappe).

Per rappresentare la struttura dell'albero, la nostra classe `TreeMap` usa un esemplare di una sottoclasse della classe `LinkedBinaryTree` vista nel Paragrafo 8.3.1. In questa implementazione abbiamo scelto di rappresentare l'albero di ricerca come un albero binario *proprio*, con la presenza esplicita dei nodi foglia come sentinelle e voci della mappa memorizzate soltanto nei nodi interni (lasciando all'Esercizio P-11.55 il compito di progettare un'implementazione più efficiente in termini di spazio occupato in memoria).

L'algoritmo `TreeSearch` presentato nel Codice 11.1 viene implementato come metodo ricorsivo privato, `treeSearch(p , k)`, che restituisce la posizione avente una voce con chiave uguale a k oppure l'ultima posizione che ha visitato lungo il percorso di ricerca, se non ha trovato quanto cercato. Questo metodo non è utilizzato soltanto per tutte le operazioni principali della mappa, cioè `get(k)`, `put(k , v)` e `remove(k)`, ma anche per la maggior parte dei metodi della mappa ordinata, perché l'ultima posizione interna visitata durante una ricerca infruttuosa contiene la massima chiave tra quelle minori di k oppure la minima chiave tra quelle maggiori di k .

Infine, osserviamo che la nostra classe `TreeMap` è stata progettata in modo da essere usata come superclasse per altre sottoclassi che implementeranno varie forme di alberi di ricerca bilanciati. Parleremo più a fondo del bilanciamento nel Paragrafo 11.2, ma ci sono due aspetti di quei progetti che hanno un impatto significativo sul codice presentato in questo

paragrafo. Innanzitutto, il nostro campo `tree` è stato dichiarato, dal punto di vista tecnico, come esemplare della classe `BalanceableBinaryTree`, che è una sottoclasse di `LinkedBinaryTree`, anche se in questo paragrafo usiamo soltanto comportamenti ereditati. In secondo luogo il nostro codice presenta qua e là invocazioni di metodi che si presume verranno prima o poi definiti, come `rebalanceAccess`, `rebalanceInsert` e `rebalanceDelete`: in questa classe questi metodi non svolgono alcuna elaborazione, ma servono come *agganci (hook)* per successive personalizzazioni all'interno di sottoclassi.

Concludiamo con una breve guida sull'organizzazione del codice.

- Codice 11.3: Prima parte della classe `TreeMap`, con i costruttori, il metodo `size` e i metodi ausiliari `expandExternal` e `treeSearch`.
- Codice 11.4: Operazioni della mappa: `get(k)`, `put(k, v)` e `remove(k)`.
- Codice 11.5: Alcuni metodi dell'ADT mappa ordinata: `lastEntry()`, `floorEntry()` e `lowerEntry()`, oltre al metodo ausiliario `protected treeMax`.
- Codice 11.6: Il metodo `entrySet` dell'ADT mappa, per consentire la scansione di tutte le voci, e il metodo `subMap(k1, k2)` dell'ADT mappa ordinata, per la scansione di un intervallo di voci.

Codice 11.3: Prima parte della classe `TreeMap` basata su un albero di ricerca binario.

```

1  /** Un'implementazione di mappa ordinata che usa un albero di ricerca binario. */
2  public class TreeMap<K,V> extends AbstractSortedMap<K,V> {
3      // per rappresentare la struttura ad albero usiamo una sottoclasse di
4      // LinkedBinaryTree, BalanceableBinaryTree (illustrata nel Paragrafo 11.2).
5      protected BalanceableBinaryTree<K,V> tree = new BalanceableBinaryTree<>();
6
7      /** Costruisce una mappa vuota usando l'ordinamento naturale tra le chiavi. */
8      public TreeMap() {
9          super();                      // il costruttore di AbstractSortedMap
10         tree.addRoot(null);           // crea una foglia sentinella come radice
11     }
12     /** Costruisce una mappa vuota ordinando le chiavi con il comparatore dato. */
13     public TreeMap(Comparator<K> comp) {
14         super(comp);                // il costruttore di AbstractSortedMap
15         tree.addRoot(null);           // crea una foglia sentinella come radice
16     }
17     /** Restituisce il numero di voci presenti nella mappa. */
18     public int size() {
19         return (tree.size() - 1)/2;    // solo i nodi interni hanno una voce
20     }
21     /** Metodo ausiliario usato per inserire una nuova voce in una foglia. */
22     private void expandExternal(Position<Entry<K,V>> p, Entry<K,V> entry) {
23         tree.set(p, entry);          // memorizza in p la nuova voce
24         tree.addLeft(p, null);       // aggiunge nuove foglie-sentinella come figli
25         tree.addRight(p, null);
26     }
27
28     // In questo codice abbiamo omesso, per brevità, una serie di metodi protected
29     // che abbreviano alcune operazioni sull'albero binario concatenato.
30     // Ad esempio, con il seguente metodo ausiliario consentiamo
31     // l'uso dell'abbreviazione root() al posto di tree.root()
32     protected Position<Entry<K,V>> root() { return tree.root(); }
33

```

```

34  /** Restituisce la posizione nel sottoalbero di p che ha la chiave data. */
35  private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key) {
36      if (isExternal(p))
37          return p;           // chiave non trovata, restituisce la foglia terminale
38      int comp = compare(key, p.getElement());
39      if (comp == 0)
40          return p;           // chiave trovata, restituisce la sua posizione
41      else if (comp < 0)
42          return treeSearch(left(p), key); // cerca nel sottoalbero sinistro
43      else
44          return treeSearch(right(p), key); // cerca nel sottoalbero destroy
45  }

```

Codice 11.4: Operazioni principali della mappa nella classe TreeMap.

```

46  /** Restituisce il valore associato alla chiave specificata (o null). */
47  public V get(K key) throws IllegalArgumentException {
48      checkKey(key);           // può lanciare IllegalArgumentException
49      Position<Entry<K,V>> p = treeSearch(root(), key);
50      rebalanceAccess(p);     // per sottoclassi di alberi bilanciati
51      if (isExternal(p)) return null; // ricerca infruttuosa
52      return p.getElement().getValue(); // corrispondenza trovata
53  }
54  /** Associa value alla chiave data, restituendo il vecchio valore sovrascritto. */
55  public V put(K key, V value) throws IllegalArgumentException {
56      checkKey(key);           // può lanciare IllegalArgumentException
57      Entry<K,V> newEntry = new MapEntry<>(key, value);
58      Position<Entry<K,V>> p = treeSearch(root(), key);
59      if (isExternal(p)) {      // la chiave è nuova
60          expandExternal(p, newEntry);
61          rebalanceInsert(p);   // per sottoclassi di alberi bilanciati
62          return null;
63      } else {                 // sostituisce il valore esistente
64          V old = p.getElement().getValue();
65          set(p, newEntry);
66          rebalanceAccess(p);   // per sottoclassi di alberi bilanciati
67          return old;
68      }
69  }
70  /** Elimina la voce con chiave k (se esiste) e restituisce il valore associato. */
71  public V remove(K key) throws IllegalArgumentException {
72      checkKey(key);           // può lanciare IllegalArgumentException
73      Position<Entry<K,V>> p = treeSearch(root(), key);
74      if (isExternal(p)) {      // chiave non trovata
75          rebalanceAccess(p);   // per sottoclassi di alberi bilanciati
76          return null;
77      } else {
78          V old = p.getElement().getValue();
79          if (isInternal(left(p)) && isInternal(right(p))) { // entrambi i figli interni
80              Position<Entry<K,V>> replacement = treeMax(left(p));
81              set(p, replacement.getElement());
82              p = replacement;
83          } // ora al massimo uno dei figli di p è interno
84          Position<Entry<K,V>> leaf = (isExternal(left(p)) ? left(p) : right(p));
85          Position<Entry<K,V>> sib = sibling(leaf);
86          remove(leaf);
87          remove(p);            // sib risale nella posizione p
88          rebalanceDelete(p);    // per sottoclassi di alberi bilanciati

```

```

    return old;
}
}

```

Codice 11.5: Alcune operazioni della mappa ordinata nella classe TreeMap.

```

92  /** Restituisce la posizione con chiave massima nel sottoalbero di radice p. */
93  protected Position<Entry<K,V>> treeMax(Position<Entry<K,V>> p) {
94      Position<Entry<K,V>> walk = p;
95      while (isInternal(walk))
96          walk = right(walk);
97      return parent(walk);           // vogliamo il genitore della foglia
98  }
99  /** Restituisce la voce con chiave massima (o null se la mappa è vuota). */
100 public Entry<K,V> lastEntry() {
101     if (isEmpty()) return null;
102     return treeMax(root()).getElement();
103 }
104 /** Restituisce la voce con chiave massima tra quelle non maggiori di key. */
105 public Entry<K,V> floorEntry(K key) throws IllegalArgumentException {
106     checkKey(key);                  // può lanciare IllegalArgumentException
107     Position<Entry<K,V>> p = treeSearch(root(), key);
108     if (isInternal(p)) return p.getElement(); // corrispondenza esatta
109     while (!isRoot(p)) {
110         if (p == right(parent(p)))
111             return parent(p).getElement(); // il genitore di p ha la chiave che cerchiamo
112         else
113             p = parent(p);
114     }
115     return null;           // non c'è nessuna chiave non maggiore di key
116 }
117 /** Restituisce la voce con chiave massima tra quelle minori di key. */
118 public Entry<K,V> lowerEntry(K key) throws IllegalArgumentException {
119     checkKey(key);                  // può lanciare IllegalArgumentException
120     Position<Entry<K,V>> p = treeSearch(root(), key);
121     if (isInternal(p) && isInternal(left(p)))
122         return treeMax(left(p)).getElement(); // questo è il predecessore di p
123     // altrimenti la ricerca è fallita o manca il figlio sinistro
124     while (!isRoot(p)) {
125         if (p == right(parent(p)))
126             return parent(p).getElement(); // il genitore di p ha la chiave che cerchiamo
127         else
128             p = parent(p);
129     }
130     return null;           // non c'è nessuna chiave minore di key
131 }

```

Codice 11.6: Operazioni della classe TreeMap che consentono di scandire l'intera mappa o una sua porzione tra due chiavi date.

```

/** Restituisce un contenitore iterabile con tutte le voci della mappa. */
public Iterable<Entry<K,V>> entrySet() {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(size());
    for (Position<Entry<K,V>> p : tree.inorder())
        if (isInternal(p)) buffer.add(p.getElement());
    return buffer;
}

```

```

139  /** Restituisce un contenitore di voci con chiavi in [fromKey, toKey]. */
140  public Iterable<Entry<K,V>> subMap(K fromKey, K toKey()) {
141      ArrayList<Entry<K,V>> buffer = new ArrayList<>(size());
142      if (compare(fromKey, toKey) < 0) // controlla che sia fromKey < toKey
143          subMapRecurse(fromKey, toKey, root(), buffer);
144      return buffer;
145  }
146  private void subMapRecurse(K fromKey, K toKey, Position<Entry<K,V>> p,
147                             ArrayList<Entry<K,V>> buffer) {
148      if (isInternal(p))
149          if (compare(p.getElement(), fromKey) < 0)
150              // la chiave di p è minore di fromKey, quindi le voci cercate sono a destra
151              subMapRecurse(fromKey, toKey, right(p), buffer);
152          else {
153              subMapRecurse(fromKey, toKey, left(p), buffer); // prima a sinistra
154              if (compare(p.getElement(), toKey) < 0) { // p è nell'intervallo
155                  buffer.add(p.getElement()); // quindi aggiungilo alla lista e considera...
156                  subMapRecurse(fromKey, toKey, right(p), buffer); // anche a destra
157              }
158          }
159      }

```

11.1.4 Prestazioni di un albero di ricerca binario

La Tabella 11.1 riporta un'analisi delle operazioni della nostra classe `TreeMap`. Quasi tutte le operazioni hanno un tempo d'esecuzione, nel caso peggiore, che dipende da h , cioè dall'altezza dell'albero in quel momento. Questo accade perché la maggior parte delle operazioni si basa sull'attraversamento di un percorso che scende dalla radice dell'albero e la lunghezza massima di un tale percorso all'interno di un albero è proporzionale all'altezza dell'albero stesso. Come si può facilmente osservare, la nostra implementazione delle operazioni principali della mappa (`get`, `put` e `remove`) e della maggior parte delle operazioni della mappa ordinata comincia con un'invocazione del metodo ausiliario `treeSearch`. Percorsi simili vengono seguiti quando si cerca la voce con chiave minima o massima all'interno di un sottoalbero, un obiettivo parziale che serve a trovare una voce sostitutiva durante una rimozione o la prima o ultima voce della mappa. Una scansione dell'intera mappa si può ottenere in un tempo $O(n)$ usando un attraversamento in ordine simmetrico dell'albero che la contiene, mentre si può dimostrare che l'implementazione ricorsiva del metodo `subMap` richiede un tempo d'esecuzione $O(s + h)$ nel caso peggiore, per un'invocazione che restituisca s elementi (si veda, a questo proposito, l'Esercizio C-11.34).

Tabella 11.1: Tempi d'esecuzione nel caso peggiore delle operazioni di una `TreeMap`.

Indichiamo con h l'altezza dell'albero e con s il numero di voci restituite da `subMap`.

Lo spazio di memoria utilizzato è $O(n)$, dove n è il numero di voci presenti nella mappa.

Metodo	Tempo d'esecuzione
<code>size</code> , <code>isEmpty</code>	$O(1)$
<code>get</code> , <code>put</code> , <code>remove</code>	$O(h)$
<code>firstEntry</code> , <code>lastEntry</code>	$O(h)$
<code>ceilingEntry</code> , <code>floorEntry</code> , <code>lowerEntry</code> , <code>higherEntry</code>	$O(h)$
<code>subMap</code>	$O(s + h)$
<code>entrySet</code> , <code>keySet</code> , <code>values</code>	$O(n)$

Un albero di ricerca binario T è, quindi, un'implementazione efficiente di una mappa con n voci soltanto se la sua altezza è piccola. Nel caso migliore, T ha altezza $h = \lceil \log(n+1) \rceil - 1$, dando luogo a prestazioni logaritmiche per la maggior parte delle operazioni della mappa. Nel caso peggiore, però, T ha altezza n , nel qual caso il comportamento sarebbe quello di una mappa implementata con una lista ordinata. Tale configurazione di caso peggiore si ottiene, ad esempio, inserendo nell'albero le voci con chiave in ordine crescente o decrescente, come di può vedere nella Figura 11.7.

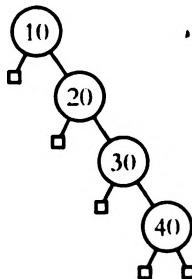


Figura 11.7: Esempio di albero di ricerca binario con altezza lineare, ottenuto inserendo le voci in base all'ordinamento crescente delle loro chiavi.

Possiamo, però, in qualche modo rassicurarci, perché, in media, un albero di ricerca binario con n voci generato da una sequenza casuale di inserimenti e rimozioni ha un'altezza attesa $O(\log n)$: la dimostrazione di questa affermazione va al di là degli obiettivi del libro, perché richiede la conoscenza del linguaggio matematico necessario per definire con precisione cosa si intenda per “sequenza casuale di inserimenti e rimozioni”, oltre a conoscenze approfondite di teoria della probabilità.

In quelle applicazioni dove non sia possibile garantire la natura casuale delle operazioni di aggiornamento, è preferibile affidarsi a varianti degli alberi di ricerca, presentati nel seguito di questo capitolo, che garantiscono un'altezza $O(\log n)$ nel caso peggiore e, così, prestazioni temporali $O(\log n)$ nel caso peggiore per ricerche, inserimenti e rimozioni.

11.2 Alberi di ricerca bilanciati

Alla fine del paragrafo precedente abbiamo osservato che, nel caso in cui si possa ipotizzare che la costruzione dell'albero di ricerca binario derivi da una sequenza casuale di inserimenti e rimozioni, il tempo d'esecuzione atteso per le principali operazioni della mappa è $O(\log n)$. Tuttavia, nel caso peggiore il tempo diventa $O(n)$, perché alcune sequenze di operazioni di aggiornamento della struttura dell'albero possono portare a un albero sbilanciato, con altezza proporzionale a n .

Nella parte restante di questo capitolo vedremo quattro algoritmi di gestione di alberi di ricerca che sono in grado di garantire prestazioni più robuste. Tre di queste strutture dati (alberi AVL, alberi *splay* e alberi rosso-nero) si basano sull'integrazione della normale operatività di un albero di ricerca binario con azioni che, quando necessario, modificano la forma dell'albero e ne riducono l'altezza.

La principale operazione che viene eseguita per ripristinare il bilanciamento di un albero di ricerca binario prende il nome di *rotazione*. Durante una rotazione, facciamo “ruotare”, appunto, un figlio al di sopra del proprio genitore, come schematizzato nella Figura 11.8.

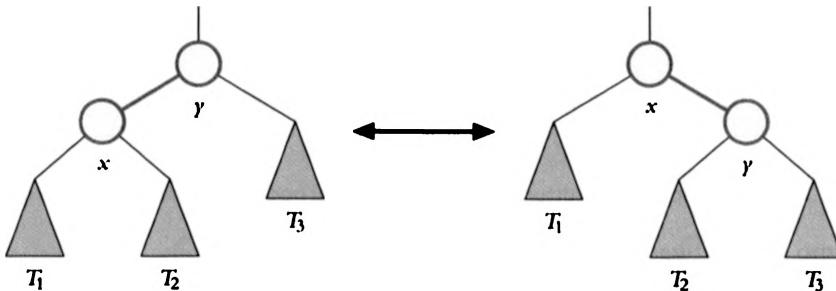


Figura 11.8: Un’operazione di rotazione in un albero di ricerca binario. La rotazione può essere eseguita per trasformare la configurazione di sinistra in quella di destra, oppure quella di destra in quella di sinistra. Si osservi che tutte le chiavi del sottoalbero T_1 , hanno chiavi minori della chiave in posizione x , tutte le chiavi del sottoalbero T_2 hanno chiavi comprese tra la chiave in posizione x e la chiave in posizione y , e tutte le chiavi del sottoalbero T_3 hanno chiavi maggiori della chiave in posizione y .

Per preservare la proprietà fondamentale dell’albero di ricerca binario durante una rotazione, osserviamo che, se la posizione x era il figlio sinistro della posizione y prima di una rotazione (e, quindi, la chiave di x era minore della chiave di y), allora y diventa il figlio destro di x dopo la rotazione, e viceversa. Inoltre, dobbiamo modificare il collegamento del sottoalbero di voci aventi chiavi che sono comprese tra le chiavi delle due posizioni che partecipano alla rotazione. Ad esempio, nella Figura 11.8 il sottoalbero T_2 contiene voci le cui chiavi sono maggiori di quella della posizione x e minori di quella della posizione y . Nella prima configurazione di quella figura, T_2 è il sottoalbero destro della posizione x , mentre nella seconda configurazione è il sottoalbero sinistro della posizione y .

Dato che una singola rotazione modifica un numero costante di relazioni genitore-figlio, se si usa una rappresentazione concatenata dell’albero binario può essere implementata in un tempo $O(1)$.

Nell’ambito di un algoritmo di bilanciamento di un albero, una rotazione consente di modificare la forma dell’albero stesso preservando la proprietà fondamentale dell’albero di ricerca. Se usata intelligentemente, questa operazione può evitare configurazioni sbilanciate dell’albero. Ad esempio, una rotazione verso destra, a partire dalla prima configurazione della Figura 11.8 per giungere alla seconda, riduce di un’unità la profondità di ciascun nodo del sottoalbero T_1 , aumentando di un’unità la profondità di ciascun nodo del sottoalbero T_3 (si noti che, invece, la profondità dei nodi del sottoalbero T_2 non è influenzata dalla rotazione).

Una o più rotazioni possono combinarsi per effettuare un più ampio ripristino del bilanciamento dell’albero. Una di tali operazioni composte che prendiamo in esame è la *ristrutturazione di una terza di nodi* (*trinode restructuring*), nella quale si considera una posizione x , il suo genitore y e il genitore del genitore, z . L’obiettivo è quello di ristrutturare il sottoalbero avente radice z in modo da ridurre la lunghezza complessiva del percorso che porta a x e ai suoi sottoalberi. Il Codice 11.7 riporta lo pseudocodice che descrive il metodo

`restructure(x)`, la cui azione è illustrata nella Figura 11.9. Nel descrivere una ristrutturazione, rinominiamo temporaneamente le posizioni x, y e z come a, b e c , in modo che a preceda b e b preceda c nell'attraversamento in ordine simmetrico di T . Come si può vedere nella Figura 11.9, le possibili corrispondenze tra la terna x, y e z e la terna a, b e c sono quattro e vengono riunificate in un unico caso tramite l'etichettatura che abbiamo definito. La ristrutturazione sostituisce z con il nodo identificato come b , facendo in modo che i figli di tale nodo diventino a e c ; inoltre, fa in modo che i figli di a e c siano i quattro precedenti figli di x, y e z (diversi da x e y), preservando la relazione di ordinamento simmetrico di tutti i nodi di T .

Codice 11.7: L'operazione di ristrutturazione di una terna di nodi in un albero di ricerca binario.

Algoritmo `restructure(x)`:

Input: Una posizione x in un albero di ricerca binario T che ha sia il genitore y sia il genitore del genitore, z

Output: L'albero T dopo la ristrutturazione (che corrisponde a una rotazione singola o doppia) che coinvolge le posizioni x, y e z

1. Sia (a, b, c) un elenco ordinato da sinistra a destra (cioè rispettando l'attraversamento in ordine simmetrico) delle posizioni x, y e z , e sia (T_1, T_2, T_3, T_4) un elenco analogamente ordinato da sinistra a destra dei quattro sottoalberi di x, y e z che non hanno radice in x, y o z .
2. Sostituire il sottoalbero avente radice z con un nuovo sottoalbero avente radice b .
3. Sia a il figlio sinistro di b e siano, rispettivamente, T_1 e T_2 i sottoalberi sinistro e destro di a .
4. Sia c il figlio destro di b e siano, rispettivamente, T_3 e T_4 i sottoalberi sinistro e destro di c .

In pratica, le modifiche di un albero T provocate da un'operazione di ristrutturazione di una terna possono essere implementate mediante una rotazione singola (come nelle Figure 11.9a e 11.9b) o una rotazione doppia (come nelle Figure 11.9c e 11.9d), dipendentemente dal caso che si presenta. Si usa una rotazione doppia quando la posizione x contiene la chiave intermedia tra le tre chiavi in esame e viene prima ruotata al di sopra del proprio genitore, poi al di sopra della posizione che originariamente era il genitore del proprio genitore. In ogni caso, la ristrutturazione viene portata a termine in un tempo $O(1)$.

11.2.1 Infrastruttura Java per bilanciare alberi di ricerca

La nostra classe `TreeMap` (presentata nel Paragrafo 11.1.3) è un'implementazione di mappa completamente funzionante, ma il tempo d'esecuzione delle sue operazioni dipende dall'altezza dell'albero e, nel caso peggiore, l'altezza può essere $O(n)$ in una mappa con n voci. Di conseguenza, abbiamo volutamente progettato la classe `TreeMap` in un modo che ci consenta di estenderla, implementando strategie più avanzate, con bilanciamento dell'albero. Negli ultimi paragrafi di questo capitolo implemeneteremo le sue sottoclassi `AVLTreeMap`, `SplayTreeMap` e `RBTreeMap`, mentre in questo paragrafo descriviamo tre forme importanti di supporto che viengono offerte dalla classe `TreeMap` alle sue sottoclassi.

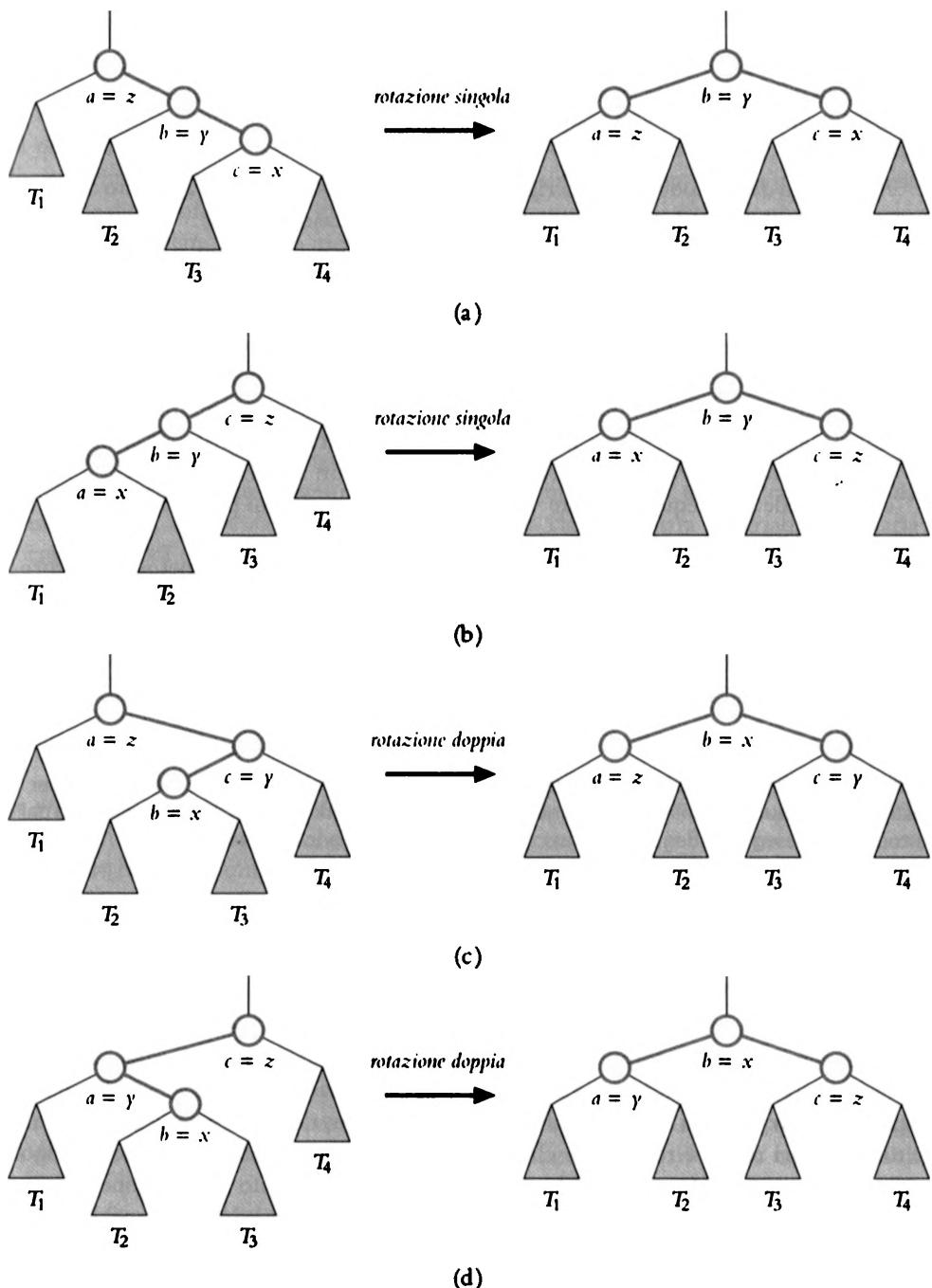


Figura 11.9: Rappresentazione schematica di un'operazione di ristrutturazione di una terna di nodi: (a e b): serve una rotazione singola; (c e d) serve una rotazione doppia.

Agganci per operazioni di ripristino del bilanciamento

La nostra implementazione delle operazioni fondamentali della mappa, nel Paragrafo 11.1.3, contiene strategicamente le invocazioni di tre metodi non pubblici che servono come *agganci* per gli algoritmi di ripristino del bilanciamento:

- All'interno del metodo `put`, nella riga 61 del Codice 11.4, dopo aver aggiunto all'albero un nuovo nodo in posizione p , c'è l'invocazione `rebalanceInsert(p)`.
- All'interno del metodo `remove`, nella riga 88 del Codice 11.4, dopo aver eliminato un nodo dall'albero, c'è l'invocazione `rebalanceDelete(p)` e p è la posizione che identifica il figlio del nodo eliminato che ha preso il suo posto.
- All'interno dei metodi `get`, `put` e `remove`, in tutti quei casi in cui *non* avvengono modifiche strutturali, c'è l'invocazione `rebalanceAccess(p)`. La posizione p , che può essere interna o esterna, rappresenta il nodo più profondo che è stato visitato durante l'operazione. Questo aggancio viene usato, in particolare, dalla struttura chiamata *albero splay* (che sarà presentata nel Paragrafo 11.4) per ristrutturare un albero in modo che i nodi a cui si accede più frequentemente vengano portati in posizioni più vicine alla radice.

Nella nostra classe `TreeMap` abbiamo definito in modo banale questi tre metodi, con corpo vuoto, nel Codice 11.8. Una sottoclasse di `TreeMap` può sovrascrivere questi metodi (anche soltanto alcuni) fornendone un'implementazione non banale, che compia azioni aventi l'obiettivo di ripristinare il bilanciamento dell'albero. Si tratta di un nuovo esempio di *schema di progettazione mediante modelli di metodi (template method design pattern)*, già discusso nel Paragrafo 2.3.3.

Codice 11.8: Definizioni banali, nella classe `TreeMap`, dei metodi che servono come agganci per la nostra infrastruttura di ripristino del bilanciamento. Questi metodi possono essere sovrascritti dalle sottoclassi per eseguire adeguate operazioni di ripristino del bilanciamento.

```
protected void rebalanceInsert(Position<Entry<K,V>> p) { }
protected void rebalanceDelete(Position<Entry<K,V>> p) { }
protected void rebalanceAccess(Position<Entry<K,V>> p) { }
```

Metodi `protected` per la rotazione e la ristrutturazione

Per consentire le operazioni di ristrutturazione, la nostra classe `TreeMap` si basa sulla memorizzazione dell'albero come esemplare di una nuova classe annidata, `BalanceableBinaryTree` (definita nel Codice 11.9 e 11.10), che è una sottoclasse della classe `LinkedBinaryTree` originariamente definita nel Paragrafo 8.3.1. Questa nuova classe mette a disposizione i metodi ausiliari `rotate` e `restructure` che, rispettivamente, eseguono una rotazione singola e una ristrutturazione (operazioni descritte all'inizio del Paragrafo 11.2). Sebbene questi metodi non vengano invocati dalle normali operazioni di `TreeMap`, la loro presenza favorisce il riutilizzo del codice, perché sono così disponibili per tutte le sottoclassi che definiscono alberi bilanciati.

Questi metodi sono implementati nel Codice 11.10. Per semplificare tale codice, abbiamo definito un ulteriore metodo ausiliario, `relink`, che collega in modo corretto due nodi, uno dei quali debba essere figlio dell'altro. L'obiettivo principale del metodo `rotate` diventa, così, quello di ridefinire le relazioni esistenti tra genitori e figli, ricollegando il nodo sottoposto a rotazione direttamente al genitore del suo genitore originale e facendo

“scivolare” il sottoalbero intermedio (quello etichettato con T_2 nella Figura 11.8) tra i nodi ruotati.

Per la ristrutturazione di una terna di nodi, per prima cosa determiniamo se serve una rotazione singola o doppia, in base a quanto descritto nella Figura 11.9. I quattro casi di quella figura mostrano un percorso discendente da z a x , attraverso y , che effettua gli spostamenti, rispettivamente, destra-destra, sinistra-sinistra, destra-sinistra e sinistra-destra. Le prime due configurazioni, in cui le due direzioni sono uguali, richiedono una rotazione singola per spostare y in alto, mentre le altre due configurazioni, con direzioni opposte tra loro, richiedono una rotazione doppia, che sposta in alto x .

Nodi specializzati, con una variabile di esemplare aggiuntiva

Molte strategie di bilanciamento degli alberi necessitano di una qualche forma di informazione “di bilanciamento” ausiliaria, da memorizzare nei nodi dell’albero. Per agevolare il compito delle sottoclassi che rappresentano alberi bilanciati, abbiamo scelto nella classe `BalanceableSearchTree` di aggiungere a ciascun nodo una variabile ausiliaria, un numero intero. Questo obiettivo è stato raggiunto definendo una nuova classe, `BSTNode`, che eredita dalla classe annidata `LinkedBinaryTree.Node`. Questa nuova classe dichiara la variabile ausiliaria e fornisce metodi per ispezionarne e modificarne il valore.

Vogliamo, poi, porre l’attenzione su un importante dettaglio di questo progetto. Ogni volta che un’operazione di basso livello che agisca sull’albero sottostante ha bisogno di un nuovo nodo, dobbiamo essere sicuri che venga creato un nodo del tipo corretto. Questo significa, nel caso del nostro albero bilanciabile, che ogni nodo deve essere di tipo `BSTNode`, con il suo campo ausiliario. La creazione di nodi, però, avviene durante operazioni di basso livello, come `addLeft` e `addRight`, che sono state definite nella classe `LinkedBinaryTree`.

Per risolvere questo problema sfruttiamo una tecnica che prende il nome di *schema di progettazione mediante metodo fabbrica (factory method design pattern)*. La classe `LinkedBinaryTree` definisce (nelle righe che vanno da 30 a 33 nel Codice 8.8) un metodo `protected, createNode`, che ha il compito di creare un nuovo nodo del tipo corretto. Gli altri metodi di quella classe semplicemente garantiscono che, quando serve un nuovo nodo, venga sempre invocato il metodo `createNode`.

Nella classe `LinkedBinaryTree`, il metodo `createNode` restituisce un esemplare del semplice tipo `Node`, mentre nella nostra classe `BalanceableBinaryTree` sovrascriviamo il metodo `createNode` (nelle righe che vanno da 22 a 27 del Codice 11.9) in modo che restituisca un nuovo esemplare della classe `BSTNode`. In questo modo modifichiamo a tutti gli effetti il comportamento delle operazioni di basso livello eseguite nella classe `LinkedBinaryTree`, in modo che usino esemplari della nostra classe “nodo” specializzata nel bilanciamento e, quindi, che ogni nodo dei nostri alberi bilanciati contenga le informazioni ausiliarie, nella nuova variabile.

Codice 11.9: La classe `BalanceableBinaryTree`, annidata all’interno della definizione della classe `TreeMap` (prosegue nel Codice 11.10).

```

1  /** Una sottoclasse di LinkedBinaryTree con supporto per il bilanciamento. */
2  protected static class BalanceableBinaryTree<K,V>
3      extends LinkedBinaryTree<Entry<K,V>> {
4      //----- classe BSTNode annidata -----
5      // estende la classe ereditata LinkedBinaryTree.Node
6      protected static class BSTNode<E> extends Node<E> {
7          int aux=0;

```

```
8     BSTNode(E e, Node<E> parent, Node<E> leftChild, Node<E> rightChild) {
9         super(e, parent, leftChild, rightChild);
10    }
11    public int getAux() { return aux; }
12    public void setAux(int value) { aux = value; }
13 //----- fine della classe BSTNode annidata -----
14
15 // metodi relativi al campo aux e basati su una posizione
16 public int getAux(Position<Entry<K,V>> p) {
17     ((BSTNode<Entry<K,V>>) p).getAux();
18 }
19 public void setAux(Position<Entry<K,V>> p, int value) {
20     ((BSTNode<Entry<K,V>>) p).setAux(value);
21 }
22 // Sovrascrive il metodo-fabbrica in modo che crei un BSTNode (e non un Node). */
23 protected
24 Node<Entry<K,V>> createNode(Entry<K,V> e, Node<Entry<K,V>> parent,
25                                     Node<Entry<K,V>> left, Node<Entry<K,V>> right) {
26     return new BSTNode(e, parent, left, right);
27 }
```

Codice 11.10: La classe `BalanceableBinaryTree`, annidata all'interno della definizione della classe `TreeMap` (continua dal Codice 11.9).

```

18  /** Collega un nodo genitore con un suo figlio, nella direzione specificata. */
19  private void relink(Node<Entry<K,V>> parent, Node<Entry<K,V>> child,
20                      boolean makeLeftChild) {
21      child.setParent(parent);
22      if (makeLeftChild)
23          parent.setLeft(child);
24      else
25          parent.setRight(child);
26  }
27  /** Ruota la Position p attorno al proprio genitore. */
28  public void rotate(Position<Entry<K,V>> p) {
29      Node<Entry<K,V>> x = validate(p);
30      Node<Entry<K,V>> y = x.getParent(); // ipotizziamo che questo nodo esista
31      Node<Entry<K,V>> z = y.getParent(); // genitore del genitore (forse null)
32      if (z == null) {
33          root = x;                                // x diventa radice dell'albero
34          x.setParent(null);
35      } else
36          relink(z, x, y == z.getLeft()); // x diventa figlio di z
37          // ora ruotiamo x e y, trasferendo il sottoalbero intermedio
38          if (x == y.getLeft()) {
39              relink(y, x.getRight(), true); // il figlio destro di x diventa sinistro di y
40              relink(x, y, false);           // y diventa figlio destro di x
41          } else {
42              relink(y, x.getLeft(), false); // il figlio sinistro di x diventa destro di y
43              relink(x, y, true);          // y diventa figlio sinistro di x
44          }
45  }
46  /** Ristruttura x insieme ai suoi due antenati più prossimi. */
47  public Position<Entry<K,V>> restructure(Position<Entry<K,V>> x) {
48      Position<Entry<K,V>> y = parent(x);
49      Position<Entry<K,V>> z = parent(y);
50      if ((x == right(y)) == (y == right(z))) { // direzioni che corrispondono
51          rotate(y);                            // rotazione singola di y
52      } else {
53          Node<Entry<K,V>> w = parent(z);
54          if (w == null) {
55              root = y;
56              y.setParent(null);
57          } else {
58              if (z == w.getLeft())
59                  relink(w, z, true);
60              else
61                  relink(w, z, false);
62          }
63          if (y == z.getLeft())
64              relink(z, y, true);
65          else
66              relink(z, y, false);
67          if (x == y.getLeft())
68              relink(y, x, true);
69          else
70              relink(y, x, false);
71      }
72  }

```

```

62     return y;
63 } else {
64     rotate(x);
65     rotate(x);
66     return x;
67 }
68 }
69 }
```

// y è la nuova radice del sottoalbero
 // direzioni diverse
 // rotazione doppia di x

// x è la nuova radice del sottoalbero

11.3 Alberi AVL

La classe `TreeMap`, che usa un albero di ricerca binario standard come struttura dati interna, potrebbe essere un'implementazione di mappa efficiente, ma le sue prestazioni di caso peggiore per le varie operazioni sono lineari, dal momento che esistono sequenze di operazioni di modifica che producono alberi con altezza lineare. In questo paragrafo descriveremo una semplice strategia di bilanciamento che garantisce tempi d'esecuzione logaritmici nel caso peggiore per tutte le operazioni fondamentali della mappa.

Definizione di albero AVL

La semplice correzione da apportare rispetto all'albero di ricerca binario standard consiste in una regola aggiuntiva, che manterrà logaritmica l'altezza dell'albero. Ricordiamo che abbiamo definito l'altezza di un sottoalbero avente radice nella posizione p di un albero come il numero di *rami* del più lungo percorso che scenda da p a una foglia (si veda il Paragrafo 8.1.3). In base a questa definizione, una foglia ha altezza 0.

In questo paragrafo usiamo la seguente *proprietà di bilanciamento in altezza*, che caratterizza la struttura di un albero di ricerca binario T in termini delle altezze dei suoi nodi.

Proprietà di bilanciamento in altezza: Per ogni posizione interna p di T , le altezze dei figli di p devono differire al massimo di un'unità.

Un albero binario T che soddisfi la proprietà di bilanciamento in altezza si chiama **albero AVL**, dalle iniziali dei suoi inventori: Adel'son-Vel'skii e Landis. La Figura 11.10 mostra un esempio di albero AVL.

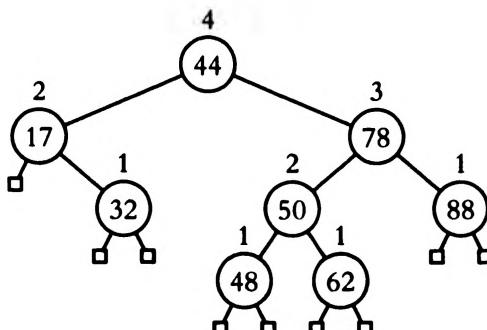


Figura 11.10: Un esempio di albero AVL. Le chiavi delle voci sono riportate all'interno dei nodi, mentre l'altezza di ciascun nodo è indicata al di sopra del nodo stesso (tutte le foglie hanno altezza 0, che non è riportata nella figura).

Una conseguenza immediata della proprietà di bilanciamento in altezza è il fatto che un sottoalbero di un albero AVL è anch'esso un albero AVL. La proprietà di bilanciamento in altezza ha anche l'importante conseguenza di mantenere bassa l'altezza dell'albero, come evidenziato dalla proposizione seguente.

Proposizione 11.1: *L'altezza di un albero AVL che memorizza n voci è $O(\log n)$.*

Dimostrazione: Invece di cercare di trovare un limite superiore per l'altezza di un albero AVL in modo diretto, si rivela più semplice lavorare sul "problema inverso" di trovare un limite inferiore al numero minimo di nodi *interni*, indicato con $n(h)$, di un albero AVL avente altezza h . Dimostreremo che $n(h)$ cresce almeno esponenzialmente: da questo, sarà facile ricavare che l'altezza di un albero AVL che memorizza n voci è $O(\log n)$.

Iniziamo osservando che $n(1) = 1$ e che $n(2) = 2$, perché un albero AVL di altezza 1 deve avere esattamente un nodo interno, mentre un albero AVL di altezza 2 deve avere almeno due nodi interni. Ora, un albero AVL con il minimo numero di nodi interni e altezza h , con $h \geq 3$, è tale che entrambi i suoi sottoalberi sono alberi AVL con il minimo numero di nodi interni: uno di altezza $h - 1$ e l'altro con altezza $h - 2$. Contando anche la radice, otteniamo la formula seguente, che mette in relazione $n(h)$ con $n(h - 1)$ e $n(h - 2)$, per $h \geq 3$:

$$n(h) = 1 + n(h - 1) + n(h - 2). \quad (11.1)$$

A questo punto, un lettore che abbia familiarità con le proprietà delle progressioni di Fibonacci (viste nei Paragrafi 2.2.3 e 5.5) avrà già riconosciuto che $n(h)$ è una funzione esponenziale di h . Per formalizzare questa osservazione, procediamo in questo modo.

La formula 11.1 implica che $n(h)$ sia una funzione strettamente crescente di h , quindi sappiamo che $n(h - 1) > n(h - 2)$. Sostituendo $n(h - 1)$ con $n(h - 2)$ nella formula 11.1 e ignorando l'addendo 1, otteniamo, per $h \geq 3$:

$$n(h) > 2 \cdot n(h - 2). \quad (11.2)$$

La formula 11.2 indica che $n(h)$ quantomeno raddoppia ogni volta che h aumenta di 2 unità, cosa che, almeno intuitivamente, fa crescere $n(h)$ esponenzialmente. Per dimostrarlo in modo formale, applichiamo la formula 11.2 più volte, ottenendo la seguente sequenza di disuguaglianze:

$$\begin{aligned} n(h) &> 2 \cdot n(h - 2) \\ &> 4 \cdot n(h - 4) \\ &> 8 \cdot n(h - 6) \\ &\vdots \\ &> 2^i \cdot n(h - 2i). \end{aligned} \quad (11.3)$$

Quindi, $n(h) > 2^i \cdot n(h - 2i)$, per qualsiasi numero intero i tale che sia $h - 2i \geq 1$. Dato che conosciamo già i valori di $n(1)$ e $n(2)$, scegliamo i in modo che $h - 2i$ sia uguale a 1 o 2. Scegliamo:

$$i = \left\lceil \frac{h}{2} \right\rceil - 1$$

Sostituendo questo valore di i nella formula 11.3, otteniamo, per $h \geq 3$:

$$\begin{aligned} n(h) &> 2^{\left\lceil \frac{h}{2} \right\rceil - 1} \cdot n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1} \cdot n(1) \\ &\geq 2^{\frac{h}{2} - 1} \end{aligned} \tag{11.4}$$

Prendendo il logaritmo di entrambi i membri della formula 11.4, otteniamo:

$$\log(n(h)) > \frac{h}{2} - 1$$

da cui:

$$h < 2\log(n(h)) + 2, \tag{11.5}$$

disegualanza che implica che un albero AVL che memorizza n voci ha un'altezza inferiore a $2 \log n + 2$. ■

Per la Proposizione 11.1 e per l'analisi che abbiamo svolto nel Paragrafo 11.1 sulle prestazioni degli alberi di ricerca binari, l'operazione `get`, in una mappa implementata con un albero AVL, viene eseguita in un tempo $O(\log n)$, dove n è il numero di voci presenti nella mappa. Ovviamente non abbiamo ancora visto come si possa preservare la proprietà di bilanciamento in altezza dopo un inserimento o una rimozione.

11.3.1 Operazioni di modifica

Dato un albero di ricerca binario T , diciamo che una sua posizione è *bilanciata (balanced)* se il valore assoluto della differenza tra le altezze dei suoi figli è al massimo uguale a un'unità, altrimenti diciamo che è *sbilanciata (unbalanced)*. Quindi, la proprietà di bilanciamento in altezza che caratterizza un albero AVL equivale a dire che tutte le posizioni dell'albero devono essere bilanciate.

Le operazioni di inserimento e rimozione per gli alberi AVL iniziano in modo simile alle corrispondenti operazioni degli alberi di ricerca binari standard, ma, per ogni operazione, è necessaria una successiva ulteriore elaborazione per ripristinare il bilanciamento delle porzioni dell'albero che sono state colpite in modo negativo dalle modifiche apportate.

Inserimento

Supponiamo che, prima dell'inserimento di una nuova voce, l'albero T soddisfi la proprietà di bilanciamento in altezza e, quindi, sia un albero AVL. L'inserimento di una nuova voce in un albero di ricerca binario, secondo la descrizione data nel Paragrafo 11.1.2, trasforma

una foglia p in un nodo interno, con due nuove foglie come figli. Questa azione può violare la proprietà di bilanciamento in altezza (come, ad esempio, nella Figura 11.11a), anche se le uniche posizioni che possono diventare sbilanciate sono gli antenati di p , perché sono le uniche posizioni i cui sottoalberi sono cambiati. Quindi, vediamo ora di descrivere come si possa ristrutturare T per eliminare qualsiasi sbilanciamento che possa essere avvenuto.

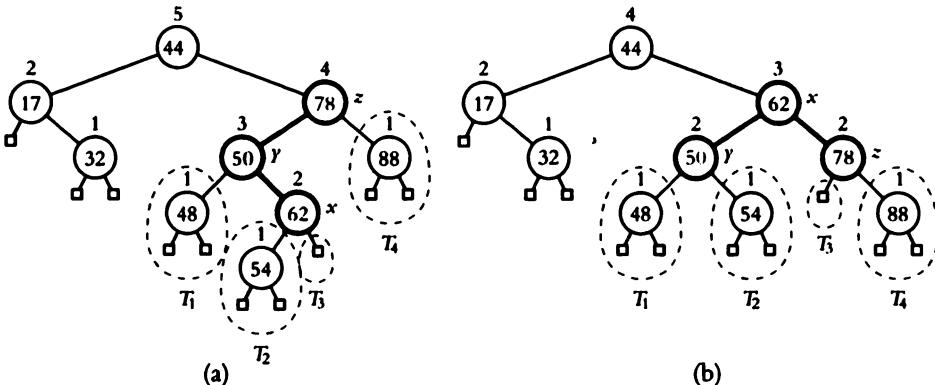


Figura 11.11: Un esempio di inserimento di una voce (con chiave 54) nell'albero AVL della Figura 11.10: (a) dopo aver aggiunto un nuovo nodo per la chiave 54, i nodi che contengono le chiavi 78 e 44 diventano sbilanciati; (b) la ristrutturazione di una terna ripristina la proprietà di bilanciamento in altezza. Sopra ogni nodo abbiamo riportato la sua altezza e abbiamo identificato gli elementi che partecipano alla ristrutturazione (i nodi x , y e z , e i sottoalberi T_1 , T_2 , T_3 e T_4).

Ripristiniamo il bilanciamento dei nodi dell'albero di ricerca binario T con una semplice strategia “cerca e ripara”. In particolare, con riferimento alla Figura 11.11a, sia z la prima posizione sbilanciata che incontriamo risalendo da p verso la radice di T . Ancora, indichiamo con y il figlio di z avente altezza maggiore (e osserviamo che y deve essere un antenato di p). Infine, sia x il figlio di y avente altezza maggiore (osservando che i due figli di y non possono avere la stessa altezza e che la posizione x deve essere anch'essa un antenato di p , eventualmente p stesso). Ripristiniamo il bilanciamento del sottoalbero avente radice z invocando il metodo di *ristrutturazione di una terza* di nodi, *restructure(x)*, già descritto nel Paragrafo 11.2. La Figura 11.11 illustra un esempio di tale ristrutturazione nel contesto di un inserimento in un albero AVL.

Per dimostrare in modo formale la correttezza di questa procedura di ripristino della proprietà di bilanciamento in altezza di un albero AVL, consideriamo cosa consegue dal fatto che z sia il più vicino antenato di p che è diventato sbilanciato dopo l'inserimento di p : l'altezza di y deve essere aumentata di un'unità per effetto dell'inserimento e ora supera di due unità l'altezza del fratello di y . Dato che la posizione y è rimasta bilanciata, prima dell'inserimento doveva avere due sottoalberi della stessa altezza e quello che ha aumentato la propria altezza di un'unità deve essere il sottoalbero che contiene x . L'altezza di tale sottoalbero è aumentata o perché x coincide con p , e quindi la sua altezza è cambiata passando da 0 a 1, oppure perché prima x aveva due sottoalberi della stessa altezza e l'altezza di quello che contiene p è aumentata di un'unità. Indicando con $h \geq 0$ l'altezza del più alto figlio di x , questo scenario può essere rappresentato come nella Figura 11.12.

Dopo la ristrutturazione, le posizioni x , y e z sono bilanciate. Inoltre, la radice del sottoalbero dopo la ristrutturazione ha altezza $h + 2$, che è precisamente l'altezza che aveva z prima dell'inserimento della nuova voce. Quindi, qualsiasi antenato di z che si fosse temporaneamente sbilanciato torna a essere bilanciato e questa unica ristrutturazione ripristina *globalmente* la proprietà di bilanciamento in altezza.

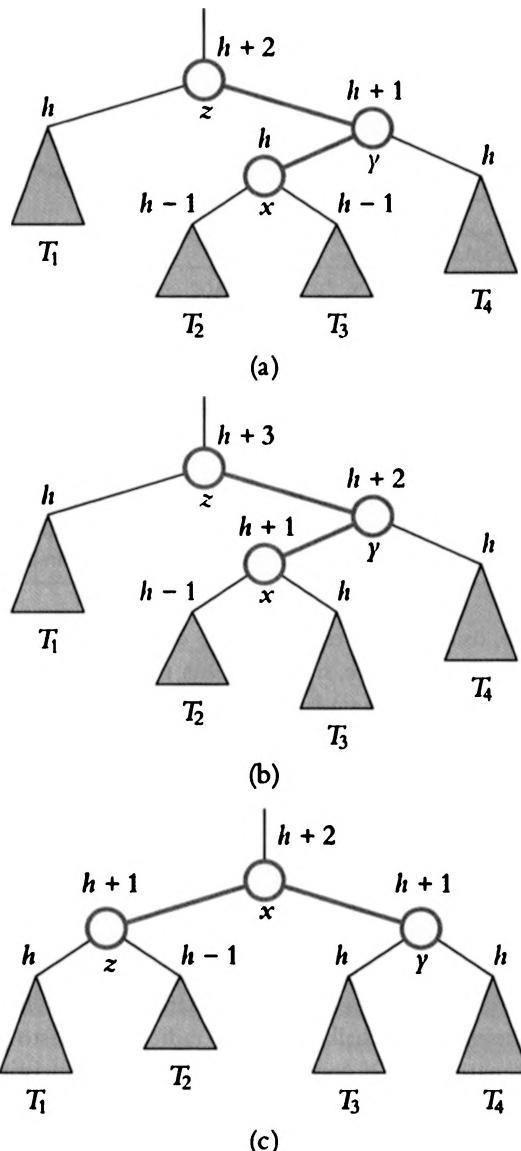


Figura 11.12: Ripristino del bilanciamento di un sottoalbero durante un tipico inserimento in un albero AVL: (a) prima dell'inserimento; (b) dopo un inserimento avvenuto nel sottoalbero T_3 che ha provocato lo sbilanciamento di z ; (c) dopo il ripristino del bilanciamento con la ristrutturazione di una terna. Si osservi che l'altezza complessiva del sottoalbero dopo l'inserimento è uguale a quella che aveva prima dell'inserimento stesso.

Rimozione

Ricordiamo che la rimozione di una voce da un albero di ricerca binario standard ha come effetto la rimozione di un nodo che non ha due figli interni. Tale modifica può violare la proprietà di bilanciamento in altezza di un albero AVL. In particolare, se la posizione p è uno dei figli (eventualmente una foglia) del nodo eliminato dall'albero T , ci può essere un nodo sbilanciato sul percorso che da p risale verso la radice di T , come si può vedere nella Figura 11.13a. In effetti, ci può essere al massimo un solo nodo sbilanciato (come si dimostrerà nell'Esercizio C-11.41).

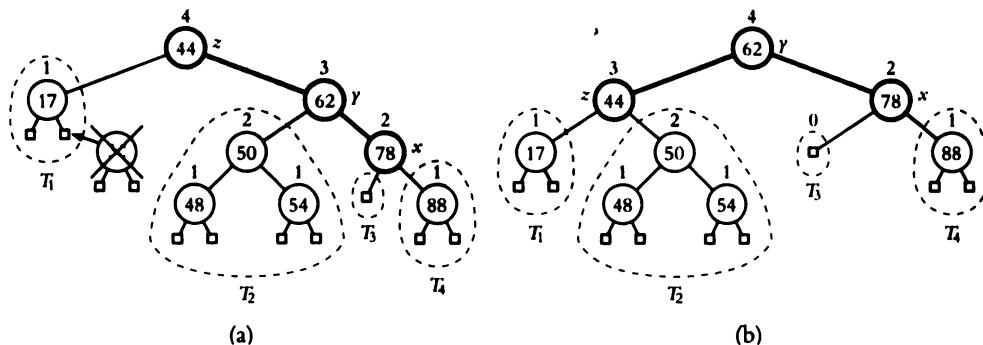


Figura 11.13: Eliminazione della voce con chiave 32 dall'albero AVL della Figura 11.11b:
 (a) dopo la rimozione del nodo contenente la chiave 32, la radice diventa sbilanciata;
 (b) la ristrutturazione di x , y e z ripristina la proprietà di bilanciamento in altezza.

Come nell'inserimento, usiamo la ristrutturazione di una terna per ripristinare il bilanciamento dell'albero T . In particolare, sia z la prima posizione sbilanciata che si incontra risalendo da p verso la radice di T e sia y il figlio di z avente altezza maggiore (y non sarà un antenato di p). Inoltre, sia x il figlio di y definito in questo modo: se uno dei figli di y è più alto dell'altro, allora x è il figlio di y più alto; altrimenti (entrambi i figli di y hanno la stessa altezza), x è il figlio di y che "si trova dalla stessa parte" di y (cioè, se y è il figlio sinistro di z , allora x sarà il figlio sinistro di y , altrimenti x sarà il figlio destro di y). Quindi, eseguiamo l'operazione `restructure(x)`, il cui risultato si può vedere nella Figura 11.13b.

Il sottoalbero ristrutturato ha come radice la posizione intermedia, indicata con b nella descrizione dell'operazione di ristrutturazione. La proprietà di bilanciamento in altezza è stata certamente ripristinata *localmente* all'interno del sottoalbero avente radice b (si veda l'Esercizio R-11.11 e l'Esercizio R-11.12). Sfortunatamente, questa ristrutturazione può ridurre di un'unità l'altezza del sottoalbero avente radice b e questo può provocare lo sbilanciamento di un antenato di b . Di conseguenza, dopo aver ripristinato il bilanciamento di z , continuiamo a risalire verso la radice di T cercando altre posizioni non bilanciate. Se ne troviamo un'altra, eseguiamo un'ulteriore operazione di ristrutturazione per ripristinare il suo bilanciamento e proseguiamo, fino a raggiungere la radice di T . Dato che, per la Proposizione 11.1, l'altezza di T è $O(\log n)$, essendo n il numero di voci, per ripristinare completamente la proprietà di bilanciamento in altezza dell'albero servono al massimo $O(\log n)$ operazioni di ristrutturazione di una terna di nodi.

Prestazioni degli alberi AVL

Per la Proposizione 11.1, l'altezza di un albero AVL con n voci è $O(\log n)$. Dato che il tempo d'esecuzione delle operazioni in un albero di ricerca binario standard è sempre limitato dall'altezza (come si può vedere nella Tabella 11.1) e dal momento che l'elaborazione aggiuntiva richiesta per gestire i fattori di bilanciamento memorizzati nei nodi dell'albero e per ristrutturare un albero AVL è al massimo proporzionale alla lunghezza di un percorso nell'albero, le operazioni tradizionali di una mappa vengono eseguite da un albero AVL in un tempo logaritmico nel caso peggiore. La Tabella 11.2 riassume questi risultati e la Figura 11.14 illustra graficamente queste prestazioni.

Tabella 11.2 Tempi d'esecuzione nel caso peggiore delle operazioni di una mappa ordinata con n voci realizzata mediante un albero AVL, con s che indica il numero di voci restituite dal metodo `subMap`.

Metodo	Tempo d'esecuzione
<code>size, isEmpty</code>	$O(1)$
<code>get, put, remove</code>	$O(\log n)$
<code>firstEntry, lastEntry</code>	$O(\log n)$
<code>ceilingEntry, floorEntry, lowerEntry, higherEntry</code>	$O(\log n)$
<code>subMap</code>	$O(s + \log n)$
<code>entrySet, keySet, values</code>	$O(n)$

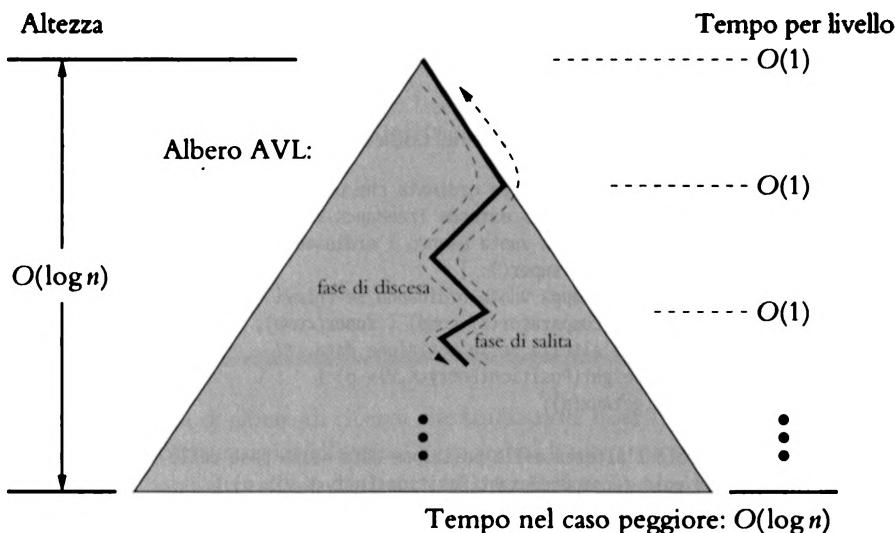


Figura 11.14: Visualizzazione del tempo d'esecuzione di ricerche e modifiche in un albero AVL. Il tempo speso per ciascun livello è $O(1)$ e si può suddividere in due fasi: una fase di discesa, che tipicamente riguarda la ricerca, e una fase di salita, che solitamente richiede l'aggiornamento delle informazioni sull'altezza dei nodi e l'esecuzione di operazioni locali di ristrutturazione di una terna (rotazioni).

11.3.2 Implementazione in Java

Nel Codice 11.11 e 11.12 presentiamo un'implementazione concreta della classe `AVLTreeMap`, che eredita dalla classe standard `TreeMap` e si basa sull'infrastruttura di bilanciamento che abbiamo descritto nel Paragrafo 11.2.1. Mettiamo qui in evidenza due aspetti importanti di questa nostra implementazione. Innanzitutto, la classe `AVLTreeMap` usa la variabile ausiliaria di ciascun nodo per memorizzare l'altezza del sottoalbero avente radice in quel nodo, mentre per le foglie tale fattore è 0. Sono inoltre presenti molti metodi ausiliari relativi alle altezze dei nodi (nel Codice 11.11).

Per implementare la logica fondamentale della strategia di bilanciamento AVL, definiamo (nel Codice 11.11) un metodo ausiliario, `rebalance`, che è in grado di ripristinare la proprietà di bilanciamento in altezza dopo un inserimento o una rimozione. Anche se i comportamenti ereditati per l'inserimento e la rimozione sono abbastanza diversi, la conseguente elaborazione necessaria per un albero AVL può essere unificata: in entrambi i casi, si tratta di risalire dalla posizione p seguendo un percorso che porta alla radice, lungo il quale avvengono le modifiche, ricalcolando l'altezza di ciascuna posizione sulla base delle altezze (aggiornate) dei suoi figli. Se troviamo una posizione sbilanciata, eseguiamo un'operazione di ristrutturazione di una terna. La marcia da p verso l'alto prosegue finché non troviamo un antenato con un'altezza che non viene modificata dall'operazione eseguita sulla mappa o che è stata riportata al suo valore precedente da un'operazione di ristrutturazione, oppure raggiungiamo la radice dell'albero (nel qual caso l'altezza complessiva dell'albero stesso sarà aumentata di un'unità). Per identificare facilmente la condizione che fa terminare la risalita, memorizziamo la "vecchia" altezza di una posizione, al suo valore prima che iniziasse l'operazione di inserimento o rimozione, e la confrontiamo con l'altezza nuova, calcolata dopo un'eventuale ristrutturazione.

Codice 11.11: La classe `AVLTreeMap` (prosegue nel Codice 11.12).

```

1  /** Un'implementazione di mappa ordinata che usa un albero AVL. */
2  public class AVLTreeMap<K,V> extends TreeMap<K,V> {
3      /** Costruisce una mappa vuota usando l'ordinamento naturale tra le chiavi. */
4      public AVLTreeMap() { super(); }
5      /** Costruisce una mappa vuota ordinando le chiavi con il comparatore dato. */
6      public AVLTreeMap(Comparator<K> comp) { super(comp); }
7      /** Restituisce l'altezza della posizione data. */
8      protected int height(Position<Entry<K,V>> p) {
9          return tree.getAux(p);
10     }
11     /** Ricalcola l'altezza della posizione data sulla base delle altezze dei figli. */
12     protected void recomputeHeight(Position<Entry<K,V>> p) {
13         tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
14     }
15     /** Restituisce true se e solo se p ha fattore di bilanciamento 1, 0 o -1. */
16     protected boolean isBalanced(Position<Entry<K,V>> p) {
17         return Math.abs(height(left(p)) - height(right(p))) <= 1;
18     }

```

Codice 11.12: La classe `AVLTreeMap` (continua dal Codice 11.11).

```

19     /** Restituisce il figlio di p che ha altezza non minore di quella dell'altro. */
20     protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
21         if (height(left(p)) > height(right(p))) return left(p); // vincitore

```

```

if (height(left(p)) < height(right(p))) return right(p); // vincitore
// figli con la stessa altezza: controlla la direzione sinistra/destra
if (isRoot(p)) return left(p); // la scelta è ininfluente
if (p == left(parent(p))) return left(p); // restituisci il figlio giusto
else return right(p);
}
*/
* Metodo ausiliario usato per ripristinare il bilanciamento dopo un'operazione di
* inserimento o rimozione. Risale il percorso da p, eseguendo una ristrutturazione
* di una terna quando trova uno sbilanciamento, continuando fino alla fine.
*/
protected void rebalance(Position<Entry<K,V>> p) {
    int oldHeight, newHeight;
    do {
        oldHeight = height(p); // non ancora ricalcolata se è un nodo interno
        if (!isBalanced(p)) { // individuato uno sbilanciamento
            // esegue una ristrutturazione, assegnando la radice risultante a
            // p e ricalcolando le nuove altezze locali dopo la ristrutturazione
            p = restructure(tallerChild(tallerChild(p)));
            recomputeHeight(left(p));
            recomputeHeight(right(p));
        }
        recomputeHeight(p);
        newHeight = height(p);
        p = parent(p);
    } while (oldHeight != newHeight && p != null);
}
/** Sovrascrive il metodo di TreeMap che viene invocato dopo un inserimento. */
protected void rebalanceInsert(Position<Entry<K,V>> p) {
    rebalance(p);
}
/** Sovrascrive il metodo di TreeMap che viene invocato dopo una rimozione. */
protected void rebalanceDelete(Position<Entry<K,V>> p) {
    if (!isRoot(p))
        rebalance(parent(p));
}
}

```

11.4 Alberi splay

La prossima struttura di albero di ricerca che studiamo è nota con il nome di *albero splay* (*splay tree*, cioè “albero esteso o allargato”) e si tratta di una struttura concettualmente abbastanza diversa dagli altri alberi di ricerca bilanciati di cui parleremo in questo capitolo, perché un albero splay non garantisce un rigido limite superiore logaritmico per l’altezza dell’albero e, in effetti, nei nodi di questo albero non viene memorizzata alcuna informazione aggiuntiva (altezza, fattore di bilanciamento o altri dati).

L’efficienza degli alberi splay è dovuta a operazioni di spostamento verso la radice, chiamate *splaying* (“estensioni”), che vengono applicate alla posizione *p* più profonda raggiunta durante un’operazione di inserimento, rimozione o anche ricerca (in pratica, si tratta di una variante della strategia euristica *move-to-front* che abbiamo visto nel Paragrafo 7.7.2 applicata alle liste). Dal punto di vista intuitivo, un’operazione di estensione fa in modo che gli elementi a cui si accede più frequentemente si trovino nelle posizioni più vicine alla radice, riducendo così i tempi di ricerca. La cosa sorprendente di questa tecnica

è che consente di garantire un tempo d'esecuzione ammortizzato logaritmico per tutte le operazioni fondamentali: inserimento, rimozione e ricerca.

11.4.1 Splaying o estensione

Dato un nodo x di un albero di ricerca binario T , eseguiamo l'*estensione* di x spostando x nella radice di T mediante una sequenza di ristrutturazioni. È importante eseguire le ristrutturazioni giuste: non è sufficiente spostare x nella radice di T mediante una sequenza di ristrutturazioni qualsiasi. La specifica operazione che va eseguita per spostare x verso l'alto dipende dalla posizione relativa di x , del suo genitore, y e del genitore del suo genitore, z (se esiste). Dovremo considerare tre casi.

zig-zig: Come nella Figura 11.15, il nodo x e il suo genitore y sono entrambi figli sinistri o entrambi figli destri. Promuoviamo x , facendo in modo che y diventi un figlio di x e z diventi un figlio di y , preservando le relazioni dei nodi di T che devono rispettare l'ordinamento derivante dall'attraversamento in ordine simmetrico.

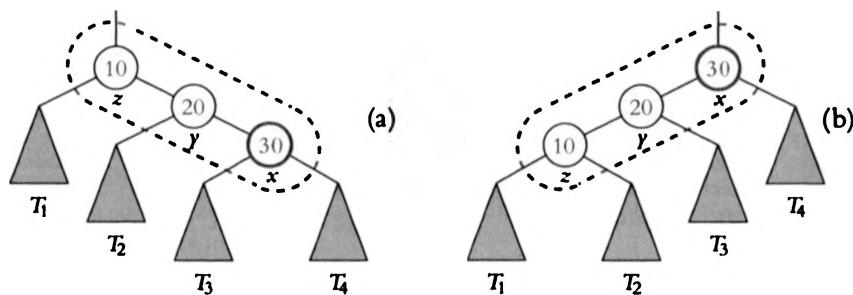


Figura 11.15: Situazione zig-zig: (a) prima; (b) dopo. C'è un'altra configurazione simmetrica, dove x e y sono entrambi figli sinistri.

zig-zag: Come nella Figura 11.16, uno dei nodi x e y è un figlio sinistro e l'altro è un figlio destro. Promuoviamo x , facendo in modo che x abbia y e z come figli, preservando le relazioni dei nodi di T che devono rispettare l'ordinamento derivante dall'attraversamento in ordine simmetrico.

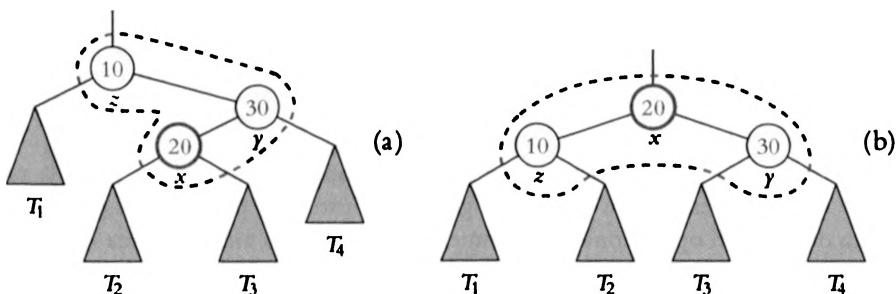


Figura 11.16: Situazione zig-zag: (a) prima; (b) dopo. C'è un'altra configurazione simmetrica, dove x è un figlio destro e y è un figlio sinistro.

zig: Come nella Figura 11.17, il genitore del nodo x non ha genitore (cioè x è figlio della radice). Eseguiamo una rotazione singola per ruotare x sopra y , facendo in modo che y diventi figlio di x , preservando le relazioni dei nodi di T che devono rispettare l'ordinamento derivante dall'attraversamento in ordine simmetrico.

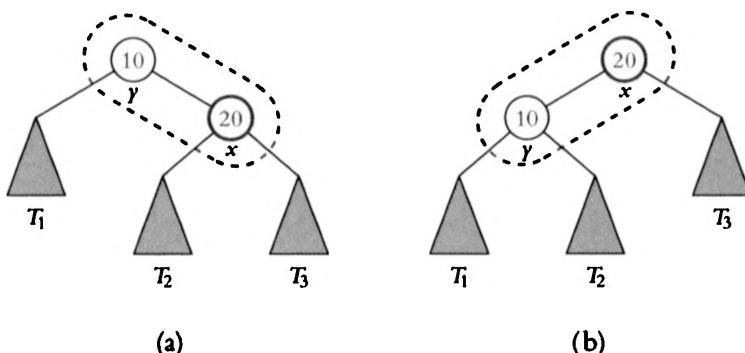


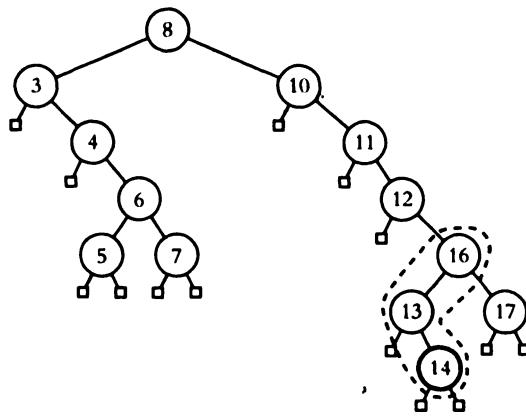
Figura 11.17: Situazione zig: (a) prima; (b) dopo. C'è un'altra configurazione simmetrica, dove x è inizialmente figlio sinistro di y .

Quando il genitore di x non è la radice eseguiamo una procedura **zig-zig** o **zig-zag**, altrimenti eseguiamo la procedura **zig**. Una fase di *estensione* consiste nella ripetizione di queste ristrutturazioni di x fino a quando la posizione x non è diventata la radice di T . Le Figure 11.18 e 11.19 mostrano, complessivamente, un esempio di estensione di un nodo.

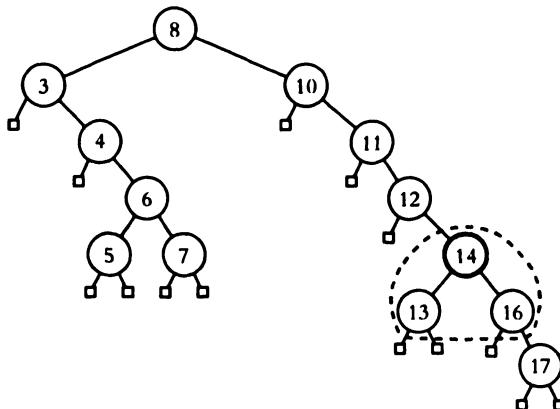
11.4.2 Quando si esegue l'estensione?

Le regole che indicano quando va eseguita una procedura di estensione sono queste:

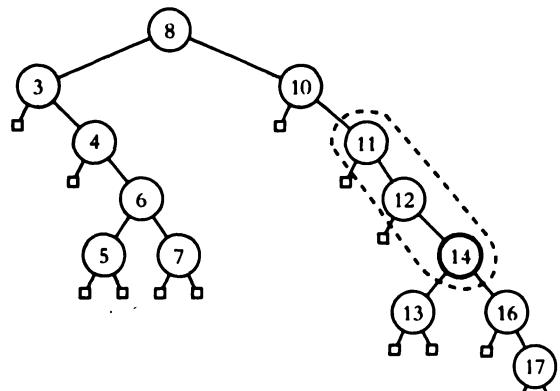
- Quando si cerca la chiave k , se k viene trovata nella posizione p , si esegue un'estensione per p , altrimenti per il genitore della foglia in cui la ricerca è terminata senza successo. Ad esempio, la procedura di estensione rappresentata nelle Figure 11.18 e 11.19 potrebbe essere successiva a una ricerca fruttuosa per la chiave 14 oppure a una ricerca infruttuosa per la chiave 15.
- Quando si inserisce la chiave k , si esegue un'estensione per il nuovo nodo interno in cui è stata inserita la chiave k . Ad esempio, la procedura di estensione rappresentata nelle Figure 11.18 e 11.19 potrebbe essere successiva all'inserimento della nuova chiave 14. Nella Figura 11.20 rappresentiamo una sequenza di inserimenti in un albero splay.



(a)

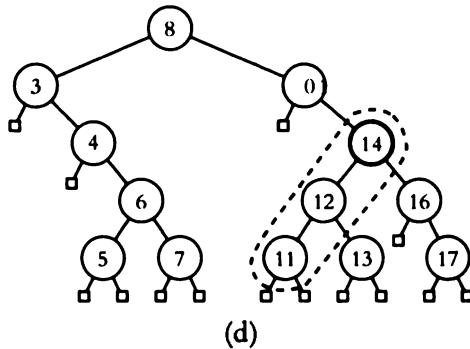


(b)

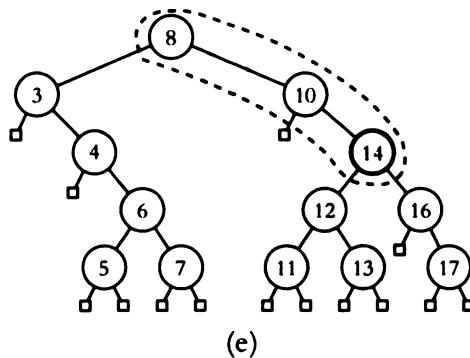


(c)

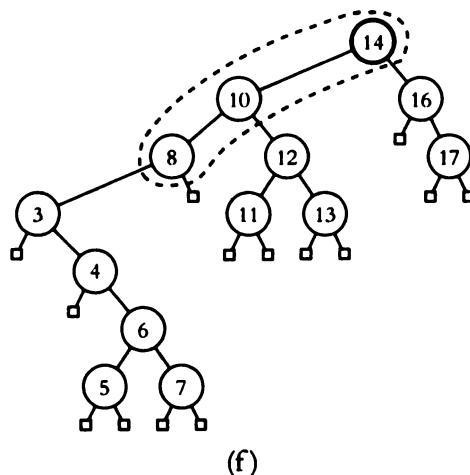
Figura 11.18: Esempio di estensione di un nodo (che prosegue nella Figura 11.19):
 (a) inizio della procedura di estensione del nodo che memorizza la chiave 14, con una fase zig-zag;
 (b) configurazione dopo il primo zig-zag; (c) il prossimo passo sarà una fase zig-zig.



(d)



(e)



(f)

Figura 11.19: Esempio di estensione di un nodo (che continua dalla Figura 11.18);
(d) configurazione dopo la fase zig-zig; (e) il prossimo passo sarà ancora una fase zig-zig;
(f) configurazione dopo la seconda fase zig-zig.

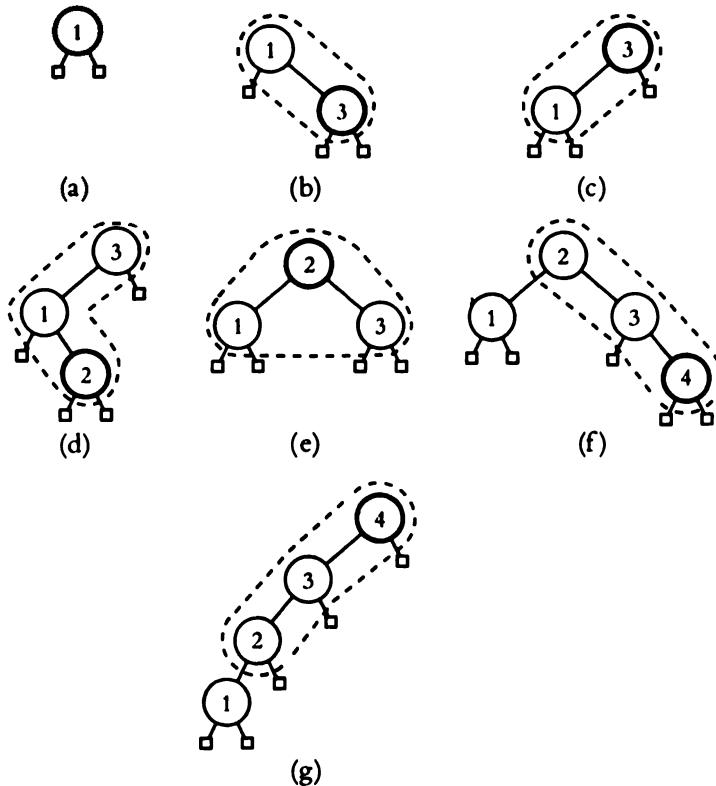


Figura 11.20: Una sequenza di inserimenti in un albero splay: (a) albero iniziale; (b) dopo l'inserimento di 3, ma prima della fase zig; (c) dopo l'estensione; (d) dopo l'inserimento di 2, ma prima della fase zig-zag; (e) dopo l'estensione; (f) dopo l'inserimento di 4, ma prima della fase zig-zig; (g) dopo l'estensione.

- Quando si elimina la chiave k , si esegue un'estensione per la posizione p , genitore del nodo rimosso; ricordiamo che, in base all'algoritmo di rimozione che abbiamo descritto per gli alberi di ricerca binari, il nodo rimosso potrebbe essere quello che originariamente conteneva la chiave k oppure un suo discendente che conteneva la chiave che è andata a sostituire k . Nella Figura 11.21 rappresentiamo un'estensione conseguente a una rimozione.

11.4.3 Implementazione in Java

Anche se l'analisi matematica delle prestazioni di un albero splay è complessa (come vedremo nel Paragrafo 11.4.4), l'*implementazione* di alberi splay è un adattamento abbastanza semplice di un albero di ricerca binario standard. Il Codice 11.13 presenta un'implementazione completa della classe `SplayTreeMap` basata sulla sottostante classe `TreeMap` e sull'uso dell'infrastruttura di bilanciamento descritta nel Paragrafo 11.2.1. Va osservato che la classe `TreeMap` originaria invoca il metodo `rebalanceAccess` non soltanto all'interno del metodo `get`, ma anche nel metodo `put`, quando modifica il valore associato a una chiave esistente, e nel metodo `remove`, quando una rimozione fallisce.

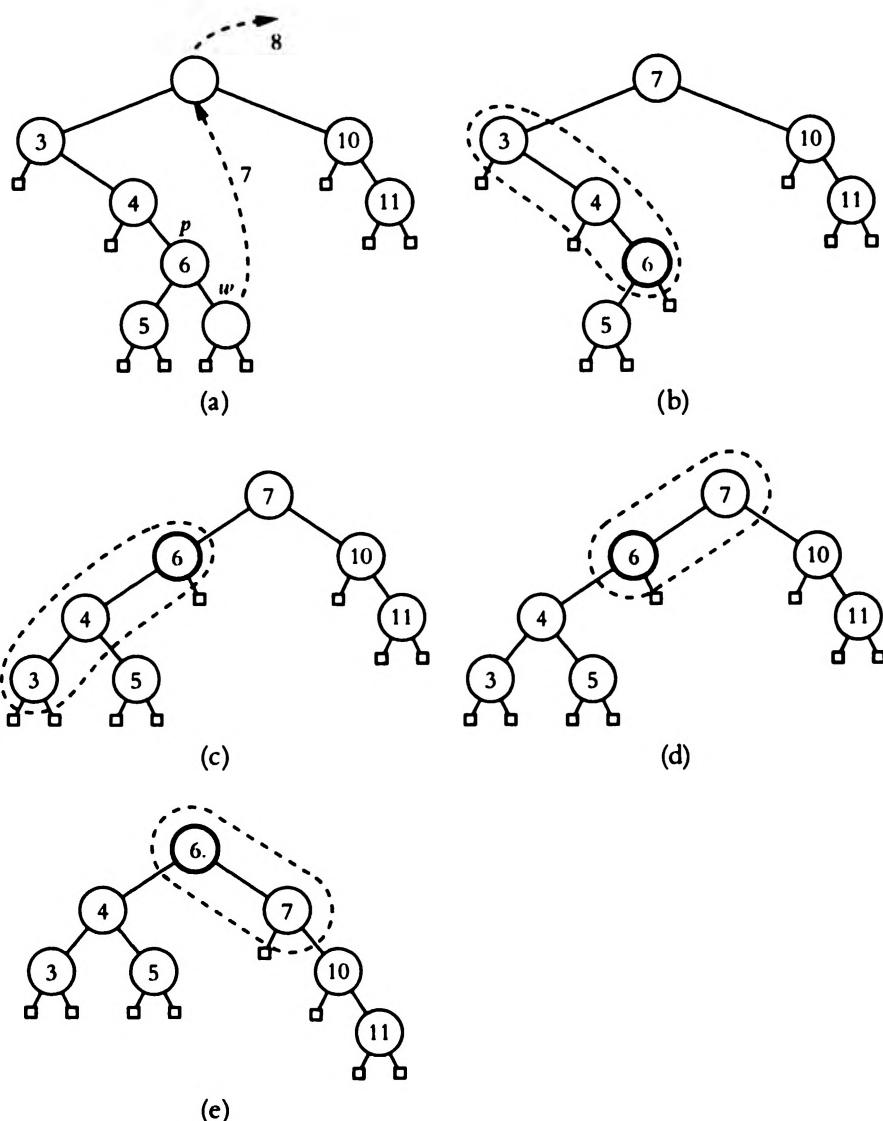


Figura 11.21: Rimozione di una chiave da un albero splay: (a) l'eliminazione della chiave 8 dal nodo radice si effettua spostando nella radice la chiave del suo predecessore w secondo l'attraversamento in ordine simmetrico, per cancellare poi w ed eseguire la procedura di estensione per il genitore p di w ; (b) la procedura di estensione per p inizia con una fase zig-zig; (c) dopo la fase zig-zig; (d) il prossimo passo è una fase zig; (e) dopo la fase zig.

Codice 11.13: Un'implementazione completa della classe SplayTreeMap.

```

1  /** Un'implementazione di mappa ordinata che usa un albero splay. */
2  public class SplayTreeMap<K,V> extends TreeMap<K,V> {
3      /** Costruisce una mappa vuota usando l'ordinamento naturale tra le chiavi. */
4      public SplayTreeMap() { super(); }
5      /** Costruisce una mappa vuota ordinando le chiavi con il comparatore dato. */
6      public SplayTreeMap(Comparator<K> comp) { super(comp); }

```

```

7  /** Metodo ausiliario per il ribilanciamento dopo un'operazione della mappa. */
8  private void splay(Position<Entry<K,V>> p) {
9      while (!isRoot(p)) {
10         Position<Entry<K,V>> parent = parent(p);
11         Position<Entry<K,V>> grand = parent(parent);
12         if (grand == null)                                     // caso zig
13             rotate(p);
14         else if ((parent == left(grand)) == (p == left(parent))) { // caso zig-zig
15             rotate(parent);          // sposta in alto parent
16             rotate(p);            // poi sposta in alto p
17         } else {                                              // caso zig-zag
18             rotate(p);          // sposta in alto p
19             rotate(p);          // sposta in alto p di nuovo
20         }
21     }
22 }
23 // sovrascriviamo i metodi di TreeMap che vengono invocati per il bilanciamento
24 protected void rebalanceAccess(Position<Entry<K,V>> p) {
25     if (isExternal(p)) p = parent(p);
26     if (p != null) splay(p);
27 }
28 protected void rebalanceInsert(Position<Entry<K,V>> p) {
29     splay(p);
30 }
31 protected void rebalanceDelete(Position<Entry<K,V>> p) {
32     if (!isRoot(p)) splay(parent(p));
33 }
34 }
```

11.4.4 Analisi ammortizzata dell'estensione o splaying*

Dopo una fase zig-zig o zig-zag, la profondità della posizione p diminuisce di due, mentre dopo una fase zig diminuisce di uno. Quindi, se p ha profondità d , un'azione di estensione completa è costituita da una sequenza di $\lfloor d/2 \rfloor$ zig-zig o zig-zag, seguita da una fase zig conclusiva se d era inizialmente dispari. Dato che una singola fase (zig-zig, zig-zag o zig) agisce su un numero costante di nodi, può essere eseguita in un tempo $O(1)$. Di conseguenza, l'estensione di una posizione p in un albero di ricerca binario richiede un tempo $O(d)$, essendo d la profondità di p in T . In altre parole, il tempo d'esecuzione dell'estensione per la posizione p è asintoticamente uguale al tempo richiesto per raggiungere la stessa posizione in una normale ricerca che proceda dalla radice di T verso il basso.

Prestazioni nel caso peggiore

Nel caso peggiore, il tempo complessivo richiesto per un'operazione di ricerca, inserimento o rimozione in un albero splay di altezza h è $O(h)$, perché la posizione che sottoponiamo a estensione può essere la posizione più profonda dell'albero. Inoltre, è possibile che h abbia lo stesso valore di n , come si può vedere nella Figura 11.20g. Quindi, dal punto di vista del caso peggiore, un albero splay non è una struttura dati molto allettante.

A dispetto delle deludenti prestazioni di caso peggiore, un albero splay si comporta bene in senso ammortizzato: in una sequenza mista di ricerche, inserimenti e rimozioni, ciascuna operazione richiede un tempo medio logaritmico, come dimostreremo a breve.

Prestazioni ammortizzate degli alberi splay

Per la nostra analisi, osserviamo che il tempo speso per eseguire una ricerca, un inserimento o una rimozione è proporzionale al tempo dell'azione di estensione conseguente, per cui consideriamo soltanto il tempo richiesto per l'estensione.

Sia T un albero splay con n chiavi e sia w un nodo di T . Definiamo la **dimensione** $n(w)$ di w come il numero di nodi che si trovano nel sottoalbero avente radice w , osservando che questa definizione implica che la dimensione di un nodo interno è superiore di un'unità rispetto alla somma delle dimensioni dei suoi figli. Definiamo, poi, il **rango** (**rank**) $r(w)$ del nodo w come il logaritmo in base 2 della dimensione di w , cioè $r(w) = \log(n(w))$. Ovviamente la radice di T ha la dimensione massima, n , e il rango massimo, $\log n$, mentre ogni foglia ha dimensione 1 e rango 0.

Usiamo, come al solito, i ciber-dollari per pagare il lavoro svolto per eseguire l'azione di estensione per una posizione p in T e ipotizziamo che un ciber-dollarino sia il costo di una fase zig, mentre ciascuna fase zig-zig o zig-zag costa due ciber-dollari. Di conseguenza, l'estensione di una posizione che si trova alla profondità d costa d ciber-dollari. Gestiremo un conto virtuale in ciber-dollari per ciascuna posizione di T al solo scopo di eseguire la nostra analisi ammortizzata e non c'è bisogno di memorizzarlo in alcun modo nella struttura dati che implementa l'albero splay T .

Un'analisi contabile dell'azione di estensione

Quando eseguiamo un'estensione, paghiamo un certo numero di ciber-dollari (il valore esatto sarà determinato al termine di questa nostra analisi) e distinguiamo tre casi:

- Se la somma pagata è uguale al lavoro svolto per l'estensione, la usiamo solamente per pagare l'estensione.
- Se la somma pagata è maggiore del lavoro svolto per l'estensione, versiamo il denaro in eccesso nei conti di alcuni nodi.
- Se la somma pagata è minore del lavoro svolto per l'estensione, preleviamo dai conti di alcuni nodi il denaro mancante.

Nel seguito dimostreremo che il pagamento di una quantità $O(\log n)$ di ciber-dollari per operazione è sufficiente per mantenere il sistema funzionante, cioè per garantire che ogni nodo abbia un saldo non negativo per il proprio conto.

Una condizione contabile invariante per l'estensione

Nel nostro schema contabile, effettuiamo trasferimenti di denaro tra i conti di nodi diversi per garantire che ci siano sempre quantità di ciber-dollari sufficienti da prelevare per pagare le estensioni quando queste servono.

Per usare il consueto metodo contabile nell'analisi dell'estensione, vogliamo preservare la seguente condizione invariante:

Prima e dopo un'estensione, ogni nodo w di T ha $r(w)$ ciber-dollari nel proprio conto.

Osserviamo che questa condizione invariante sembra sensata anche dal punto di vista finanziario, perché non richiede di fare versamenti preliminari per sovvenzionare un albero con zero chiavi.

Sia $r(T)$ la somma dei ranghi di tutti i nodi di T . Per preservare la condizione invariante dopo un'estensione, dobbiamo effettuare un pagamento uguale al lavoro svolto per l'estensione stessa, sommato alla variazione di $r(T)$. Parliamo di *sottofase* (*substep*) per indicare una singola operazione zig, zig-zig o zig-zag durante un'estensione, e indichiamo con $r(w)$ e $r'(w)$, rispettivamente, il rango del nodo w di T prima e dopo l'estensione. La proposizione seguente ci fornisce un limite superiore alla variazione di $r(T)$ provocata da una singola sottofase di un'estensione e nella nostra analisi di un'estensione completa di un nodo fino alla radice useremo ripetutamente questa affermazione.

Proposizione 11.2: *Sia δ la variazione di $r(T)$ provocata da una singola sottofase (zig, zig-zig o zig-zag) per l'estensione del nodo x di T . Valgono le seguenti diseguaglianze:*

- $\delta \leq 3(r'(x) - r(x)) - 2$ se la sottofase è zig-zig o zig-zag.
- $\delta \leq 3(r'(x) - r(x))$ se la sottofase è zig.

Dimostrazione: Useremo la seguente proprietà dei logaritmi, valida se $a > 0$, $b > 0$ e $c > a + b$:

$$\log a + \log b < 2\log c - 2. \quad (11.6)$$

Consideriamo la variazione di $r(T)$ provocata da ciascun tipo di sottofase di un'estensione.

zig-zig: Ricordando la Figura 11.15, dal momento che la dimensione di ciascun nodo è uguale a un'unità sommata alla dimensione dei suoi figli, osserviamo che durante un'operazione zig-zig cambia soltanto il rango di x , y e z , dove, come al solito, y è il genitore di x e z è il genitore di y . Ancora, $r'(x) = r(z)$, $r'(y) \leq r'(x)$ e $r(x) \leq r(y)$. Quindi:

$$\begin{aligned} \delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &= r'(y) + r'(z) - r(x) - r(y) \\ &\leq r'(x) + r'(z) - 2r(x). \end{aligned} \quad (11.7)$$

Si noti che $n(x) + n'(z) < n'(x)$. Quindi, per la 11.6, $r(x) + r'(x) < 2r'(x) - 2$, cioè:

$$r'(z) < 2r'(x) - r(x) - 2.$$

Questa diseguaglianza e la formula 11.7 implicano:

$$\begin{aligned} \delta &\leq r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \\ &\leq 3(r'(x) - r(x)) - 2. \end{aligned}$$

zig-zag: Ricordiamo la Figura 11.16 e di nuovo, per la definizione di dimensione e rango, osserviamo che durante un'operazione zig-zag cambia soltanto il rango di x , y e z , dove, come al solito, y è il genitore di x e z è il genitore di y . Ancora, $r(x) < r(y) < r(z) = r'(x)$. Quindi:

$$\begin{aligned}
 \delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
 &= r'(y) + r'(z) - r(x) - r(y) \\
 &\leq r'(y) + r'(z) - 2r(x).
 \end{aligned} \tag{11.8}$$

Si noti che $r'(y) + r'(z) < r'(x)$. Quindi, per la 11.6, $r'(y) + r'(z) < 2r'(x) - 2$, cioè:

$$\begin{aligned}
 \delta &\leq 2r'(x) - 2 - 2r(x) \\
 &= 2(r'(x) - r(x)) - 2 \leq 3(r'(x) - r(x)) - 2.
 \end{aligned}$$

zig: Ricordando la Figura 11.17, durante un'operazione zig cambia soltanto il rango di x e y , dove y è il genitore di x . Ancora, $r'(y) \leq r(y)$ e $r'(x) \geq r(x)$. Quindi:

$$\begin{aligned}
 \delta &= r'(y) + r'(x) - r(y) - r(x) \\
 &\leq r'(x) - r(x) \\
 &\leq 3(r'(x) - r(x)). \blacksquare
 \end{aligned}$$

Proposizione 11.3: Sia T un albero splay con radice t e sia Δ la variazione totale di $r(T)$ provocata dall'estensione del nodo x avente profondità d . Abbiamo:

$$\Delta \leq 3(r(t) - r(x)) - d + 2.$$

Dimostrazione: L'estensione del nodo x è costituita da $c = \lceil d/2 \rceil$ sottofasi, ciascuna delle quali è di tipo zig-zig o zig-zag, tranne eventualmente l'ultima, che è una sottofase zig quando d è dispari. Sia $r_0(x) = r(x)$ il rango iniziale di x e, per $i = 1, \dots, c$, sia $r_i(x)$ il rango di x dopo la i -esima sottofase; infine, sia δ_i la variazione di $r(T)$ provocata dalla i -esima sottofase. Per la Proposizione 11.2, la variazione totale Δ di $r(T)$ provocata dall'estensione di x è:

$$\begin{aligned}
 \Delta &= \sum_{i=1}^c \delta_i \\
 &\leq 2 + \sum_{i=1}^c 3(r_i(x) - r_{i-1}(x)) - 2 \\
 &= 3(r_c(x) - r_0(x)) - 2c + 2 \\
 &\leq 3(r(t)) - r(x)) - d + 2 \blacksquare
 \end{aligned}$$

Per la Proposizione 11.3, se per l'estensione del nodo x paghiamo una somma pari a $3(r(t) - r(x)) + 2$ ciber-dollar, disponiamo di denaro a sufficienza per preservare la condizione invariante, conservando $r(w)$ ciber-dollar in ciascun nodo w di T e pagando per intero il lavoro svolto per l'estensione, che costa d ciber-dollar. Dato che la dimensione della radice t è n , il suo rango $r(t) = \log n$ e, dal momento che $r(x) \geq 0$, la somma di denaro pagata per l'estensione è $O(\log n)$ ciber-dollar. Per completare la nostra analisi, dobbiamo soltanto calcolare il costo richiesto per preservare la condizione invariante quando un nodo viene inserito o eliminato.

Quando si inserisce un nuovo nodo w in un albero splay avente n chiavi, aumenta il rango di tutti gli antenati di w . Siano, in particolare, $w_0, \dots, w_i, \dots, w_d$ gli antenati di w , dove w_0 è w , w_i è il genitore di w_{i+1} e w_d è la radice. Per $i = 1, \dots, d$, siano $n'(w_i)$ e $n(w_i)$, rispettivamente, la dimensione di w_i prima e dopo l'inserimento, e siano $r'(w_i)$ e $r(w_i)$, rispettivamente, il rango di w_i prima e dopo l'inserimento. Ciò detto, abbiamo:

$$n'(w_i) = n(w_i) + 1.$$

Ancora, dato che $n(w_i) + 1 \leq n(w_{i+1})$, per $i = 0, 1, \dots, d - 1$, abbiamo che per ogni i in tale intervallo vale la seguente disegualanza:

$$r'(w_i) = \log(n'(w_i)) = \log(n(w_i) + 1) \leq \log(n(w_{i+1})) = r(w_{i+1}).$$

Quindi, la variazione totale di $r(T)$ provocata dall'inserimento è:

$$\begin{aligned} \sum_{i=1}^d (r'(w_i) - r(w_i)) &\leq r'(w_d) + \sum_{i=1}^{d-1} (r(w_{i+1}) - r(w_i)) \\ &= (r'(w_d) - r(w_0)) \\ &\leq \log n \end{aligned}$$

Per concludere, il pagamento di una somma pari a $O(\log n)$ ciber-dollarli è sufficiente per preservare la condizione invariante anche quando viene inserito un nuovo nodo.

Quando si cancella un nodo w da un albero splay avente n chiavi, diminuisce il rango di tutti gli antenati di w , quindi la variazione totale di $r(T)$ provocata dalla rimozione è negativa e non c'è bisogno di effettuare alcun pagamento per preservare la condizione invariante. Possiamo, quindi, riassumere la nostra analisi ammortizzata con la seguente proposizione (che a volte viene chiamata "proposizione del bilanciamento" per gli alberi splay):

Proposizione 11.4: Consideriamo una sequenza di m operazioni in un albero splay, ciascuna delle quali può essere una ricerca, un inserimento o una rimozione, a partire da un albero avente zero chiavi. Ancora, sia n_i il numero di chiavi presenti nell'albero dopo l'operazione i -esima e sia n il numero totale di inserimenti. Il tempo d'esecuzione complessivo per la sequenza di operazioni è:

$$O\left(m + \sum_{i=1}^m \log n_i\right)$$

che è una quantità $O(m \log n)$.

In altre parole, il tempo d'esecuzione ammortizzato per eseguire una ricerca, un inserimento o una rimozione in un albero splay è $O(\log n)$, dove n è la dimensione dell'albero splay al momento dell'operazione. Quindi, un albero splay è in grado di ottenere prestazioni temporali ammortizzate logaritmiche per l'implementazione di una mappa ordinata. Queste prestazioni ammortizzate corrispondono alle prestazioni di caso peggiore degli alberi AVL.

degli alberi (2, 4) e degli alberi rosso-nero, ma sono ottenute usando un semplice albero binario, che non necessita di alcuna informazione aggiuntiva memorizzata in ciascuno dei suoi nodi. Inoltre, gli alberi splay godono di parecchie altre proprietà interessanti che non si ritrovano negli altri alberi di ricerca bilanciati. Ne vedremo una nella proposizione seguente (che viene spesso chiamata "ottimalità statica" per gli alberi splay):

Proposizione 11.5: Consideriamo una sequenza di m operazioni in un albero splay, ciascuna delle quali può essere una ricerca, un inserimento o una rimozione, a partire da un albero splay T avente zero chiavi. Ancora, sia $f(i)$ il numero di accessi che riguardano la voce i (numero che viene anche detto "frequenza di accesso") e sia n il numero totale di voci presenti nell'albero. Se si accede a ogni voce almeno una volta, il tempo d'esecuzione complessivo per la sequenza di operazioni è:

$$O\left(m + \sum_{i=1}^n f(i) \log(m/f(i))\right)$$

Non dimostriamo questa affermazione, ma la dimostrazione non è così difficile come si potrebbe immaginare. L'aspetto da evidenziare è che questa proposizione afferma che il tempo d'esecuzione ammortizzato per accedere alla voce i è $O(\log(m/f(i)))$.

11.5 Alberi (2, 4)

In questo paragrafo esamineremo una struttura dati che prende il nome di *albero (2, 4)*. Si tratta di un esempio particolare di una struttura più generale nota come *albero di ricerca a più vie* (*multiway search tree*), nel quale i nodi interni possono avere più di due figli. Nel Paragrafo 15.3 vedremo altre forme di alberi a più vie.

11.5.1 Alberi di ricerca a più vie

Ricordiamo che gli alberi generici sono definiti in modo che i nodi interni possano avere molti figli. In questo paragrafo vedremo come gli alberi generici possano essere usati come alberi di ricerca a più vie: le voci di una mappa vengono memorizzate in un albero di ricerca sotto forma di coppia (k, v) , dove k è la chiave e v è il valore associato alla chiave.

Definizione di albero di ricerca a più vie

Sia w un nodo di un albero ordinato. Diciamo che w è un *d-nodo* se w ha d figli. Definiamo un albero di ricerca a più vie come un albero ordinato T che ha le seguenti proprietà, illustrate nella Figura 11.22a:

- Ogni nodo interno di T ha almeno due figli, cioè è un *d-nodo* con $d \geq 2$.
- Ogni *d-nodo* interno w di T i cui figli siano c_1, \dots, c_d , memorizza un insieme ordinato di $d - 1$ coppie chiave-valore $(k_1, v_1), \dots, (k_{d-1}, v_{d-1})$, con $k_1 \leq \dots \leq k_{d-1}$.
- Definiamo convenzionalmente $k_0 = -\infty$ e $k_d = +\infty$. Per ogni voce (k, v) memorizzata in un nodo del sottoalbero di w avente radice c_i , con $i = 1, \dots, d$, si ha $k_{i-1} \leq k \leq k_i$.

Questo significa che, se pensiamo all'insieme delle chiavi memorizzate in w come se contiene anche le finte chiavi speciali, $k_0 = -\infty$ e $k_d = +\infty$, allora una chiave k memorizzata nel sottoalbero di T avente radice nel figlio c_i deve essere "compresa" tra due delle chiavi memorizzate in w . Questo semplice punto di vista genera la regola secondo cui un d -nodo memorizza $d - 1$ chiavi "vere" e costituisce anche la base dell'algoritmo di ricerca che opera all'interno di un albero di ricerca a più vie.

In base alla definizione precedente, i nodi esterni di un albero di ricerca a più vie non memorizzano alcun dato e servono soltanto come "segnaposto": come visto per gli alberi di ricerca binari (Paragrafo 11.1), in pratica è possibile sostituirli con riferimenti `null`. Un albero di ricerca binario può, ovviamente, essere visto come caso speciale di alberi di ricerca a più vie, con ciascun nodo interno che memorizza una sola voce e ha due soli figli.

Indipendentemente, però, dal fatto che i nodi interni di un albero di ricerca a più vie abbiano due o più figli, esiste un'interessante relazione che lega il numero di coppie chiave-valore e il numero di nodi esterni presenti in un albero di ricerca a più vie.

Proposizione 11.6: *Un albero di ricerca a più vie con n voci ha $n + 1$ nodi esterni.*

Lasciamo la dimostrazione di questa affermazione all'Esercizio C-11.49.

Ricerca in un albero di ricerca a più vie

La ricerca di una voce avente chiave k all'interno di un albero di ricerca a più vie T è semplice. Si effettua la ricerca seguendo un percorso che scende in T a partire dalla radice (come si può vedere nella Figura 11.22b e c). Quando si giunge, durante la ricerca, nel d -nodo w , si confronta la chiave k con le chiavi k_1, \dots, k_{d-1} memorizzate in w . Se esiste un valore di i tale che $k = k_i$, allora la ricerca termina con successo. Altrimenti, si continua la ricerca nel figlio c_i di w tale che $k_{i-1} < k < k_i$ (ricordando che abbiamo convenzionalmente definito $k_0 = -\infty$ e $k_d = +\infty$). Se si raggiunge un nodo esterno, si sa che in T non esiste alcuna voce con chiave k e la ricerca termina senza successo.

Strutture dati per rappresentare alberi di ricerca a più vie

Nel Paragrafo 8.3.3 abbiamo visto una struttura dati concatenata adatta a rappresentare un albero generico e quella rappresentazione può essere utilizzata anche per un albero di ricerca a più vie. Quando si usa un albero generico per implementare un albero di ricerca a più vie, occorre memorizzare in ciascun nodo una o più coppie chiave-valore, associate a quel nodo: dobbiamo, cioè, memorizzare in w un riferimento a un contenitore che memorizzi le voci associate a w .

Durante la ricerca di una chiave k in un albero di ricerca a più vie, l'operazione principale che serve quando si esamina un nodo è l'individuazione della sua chiave minima che sia non minore di k . Per questa ragione, è naturale memorizzare in una mappa ordinata l'informazione associata a un nodo, perché questo consente l'uso del metodo `ceilingEntry(k)`. Diciamo solitamente che una tale mappa ricopre il ruolo di struttura dati *secondaria* per fornire supporto alla struttura dati *primaria* o principale, che rappresenta l'intero albero di ricerca a più vie. Questo modo di ragionare potrebbe, a prima vista, sembrare "circolare", perché ci serve un modo per rappresentare una mappa ordinata (secondaria) per poter rappresentare una mappa ordinata (principale). Per evitare qualsiasi dubbio di dipendenza circolare, possiamo però usare una tecnica di *bootstrap* ("che si regge in piedi da sola"), dove

una soluzione semplice di un problema serve a definire una soluzione nuova e più avanzata per lo stesso problema.

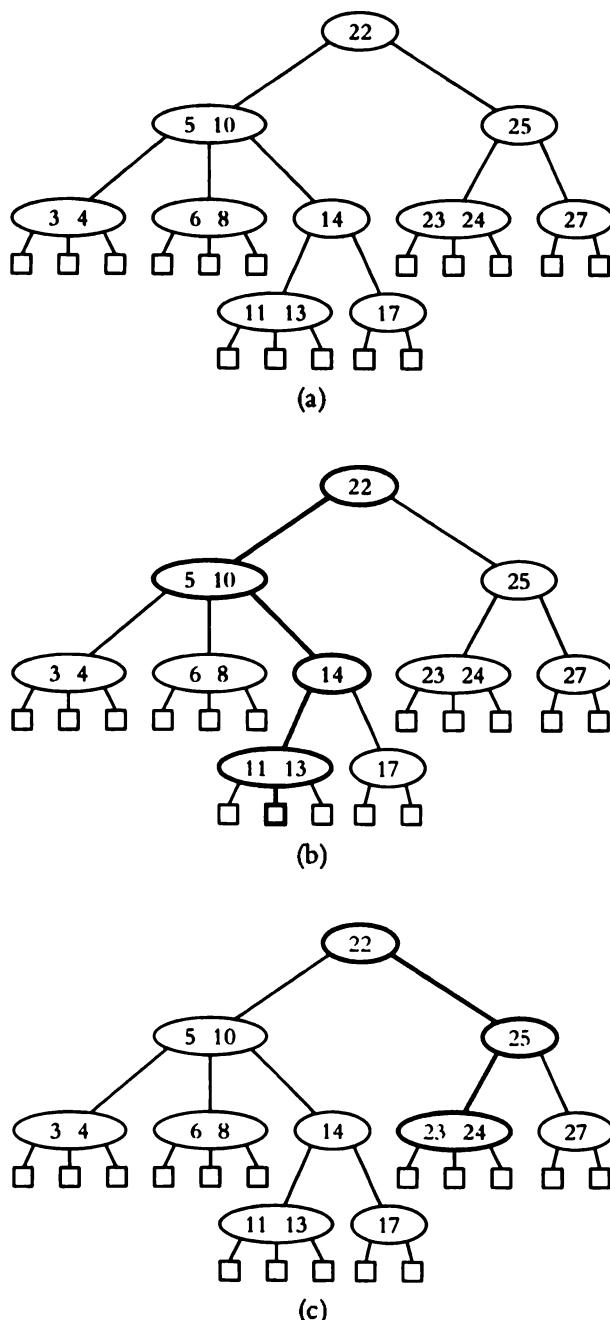


Figura 11.22: (a) Un albero di ricerca a più vie, T ; (b) il percorso di ricerca in T per la chiave 12 (una ricerca infruttuosa); (c) il percorso di ricerca in T per la chiave 24 (una ricerca fruttuosa).

Nell'ambito degli alberi di ricerca a più vie, una scelta naturale per l'implementazione delle strutture secondarie in ciascun nodo è la classe `SortedTableMap` vista nel Paragrafo 10.3.1. Se troviamo una voce corrispondente alla chiave k , vogliamo trovare il valore associato, altrimenti cerchiamo il figlio c_i tale che $k_{i-1} < k < k_i$; quindi, è bene che, nella mappa secondaria, la chiave k sia associata alla coppia (v_p, c_i) . Con una tale realizzazione dell'albero di ricerca a più vie T , l'elaborazione di un d -nodo w mentre si cerca in T una voce con chiave k può essere eseguita usando un'operazione di ricerca binaria in un tempo $O(\log d)$. Sia d_{\max} il numero massimo di figli dei nodi di T e sia h l'altezza di T : il tempo d'esecuzione di una ricerca in T è, quindi, $O(h \log d_{\max})$. Se d_{\max} è costante, tale tempo d'esecuzione diventa $O(h)$.

L'obiettivo principale per raggiungere una buona efficienza in un albero di ricerca a più vie è quello di tenere bassa l'altezza dell'albero. Vedremo, quindi, una strategia che impone a d_{\max} il limite superiore 4, garantendo che l'altezza h sia logaritmica in funzione di n (che è, come al solito, il numero totale di voci presenti nella mappa).

11.5.2 Operazioni in un albero (2, 4)

Una forma di albero di ricerca a più vie che rimane bilanciato usando piccole strutture dati secondarie in ciascun nodo è l'*albero (2, 4)*, detto anche *albero 2-4* o *albero 2-3-4*. Questa struttura dati raggiunge tali obiettivi preservando due semplici proprietà (si veda la Figura 11.23):

Vincolo sulla dimensione: Ogni nodo interno ha al massimo quattro figli.

Vincolo sulla profondità: Tutti i nodi esterni hanno la stessa profondità.

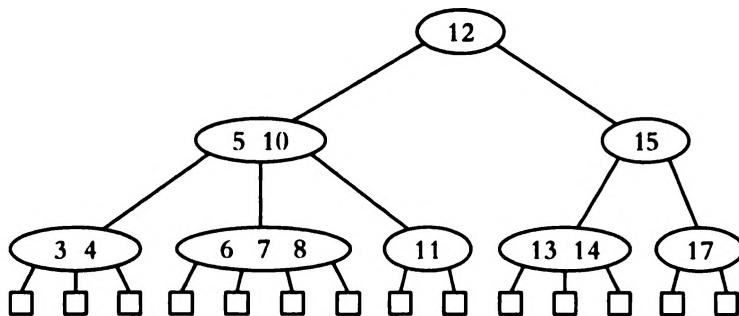


Figura 11.23: Un albero (2, 4).

Come al solito, ipotizziamo che i nodi esterni siano vuoti e, per semplicità, descriviamo i metodi di ricerca e modifica assumendo che i nodi esterni siano reali, anche se questo non è strettamente necessario.

Il vincolo sulla dimensione per gli alberi (2, 4) fa in modo che i nodi dell'albero di ricerca a più vie siano semplici e rende ragione anche del nome alternativo, “albero 2-3-4”, perché il vincolo implica che ogni nodo interno dell'albero abbia 2, 3 o 4 figli. Un'altra implicazione di questa regola è la possibilità di rappresentare la mappa secondaria memorizzata in ciascun nodo interno con una lista non ordinata o un array ordinato, pur

ottenendo prestazioni temporali $O(1)$ per tutte le operazioni (perché $d_{\max} = 4$). Il vincolo sulla profondità, d'altra parte, porta a un limite superiore per l'altezza di un albero $(2, 4)$.

Proposizione 11.7: *L'altezza di un albero $(2, 4)$ che memorizza n voci è $O(\log n)$.*

Dimostrazione: Sia h l'altezza di un albero $(2, 4)$ T che memorizza n voci. Dimostreremo l'affermazione dimostrando che:

$$\frac{1}{2} \log(n+1) \leq h \leq \log(n+1) \quad (11.9)$$

Per dimostrare questa affermazione, per prima cosa osserviamo che, per il vincolo sulla dimensione, possiamo avere al massimo 4 nodi con profondità 1, al massimo 4^2 nodi con profondità 2, e così via. Di conseguenza, il numero di nodi esterni in T è al massimo 4^h . Analogamente, per il vincolo sulla profondità e la definizione di albero $(2, 4)$, dobbiamo avere almeno 2 nodi con profondità 1, almeno 2^2 nodi con profondità 2, e così via, per cui il numero di nodi esterni in T è almeno 2^h . Inoltre, per la Proposizione 11.6, il numero di nodi esterni in T è $n + 1$. Otteniamo, quindi:

$$2^h \leq n + 1 \leq 4^h.$$

Prendendo il logaritmo in base 2 dei termini di entrambe le diseguaglianze, otteniamo che:

$$h \leq \log(n+1) \leq 2h,$$

dimostrando così, dopo aver sistemato opportunamente i vari termini, l'affermazione fatta nella formula 11.9. ■

La Proposizione 11.7 afferma che i vincoli sulla dimensione e sulla profondità sono sufficienti per tenere bilanciato un albero a più vie. Inoltre, questa proposizione implica che una ricerca in un albero $(2, 4)$ richiede un tempo $O(\log n)$ e che la scelta specifica adottata per la realizzazione delle strutture secondarie in ciascun nodo non è una scelta di progetto critica, perché il numero massimo di figli di ciascun nodo, d_{\max} , è costante.

Tuttavia, la gestione dei vincoli sulla dimensione e sulla profondità richiede qualche sforzo dopo l'esecuzione di un inserimento o di una rimozione in un albero $(2, 4)$. Queste operazioni saranno il nostro prossimo obiettivo.

Inserimento

Per inserire una nuova voce (k, v) , con chiave k , in un albero $(2, 4)$ T , per prima cosa facciamo una ricerca della chiave k . Nell'ipotesi che T non contenga alcuna voce con chiave k , questa ricerca termina senza successo in un nodo esterno z . Sia w il genitore di z : inseriamo la nuova voce nel nodo w e aggiungiamo a w un nuovo figlio y (un nodo esterno) alla sinistra di z .

Il nostro metodo di inserimento rispetta il vincolo sulla profondità, perché aggiungiamo un nodo esterno allo stesso livello di altri nodi esterni già esistenti, ma può violare il vincolo sulla dimensione. Infatti, se il nodo w aveva già 4 figli, dopo l'inserimento diventa un

5-nodo, quindi l'albero T non è più un albero (2; 4). Questo tipo di violazione del vincolo sulla dimensione è un *overflow* (*trabocco*) nel nodo w e deve essere risolta per ripristinare le proprietà dell'albero (2, 4). Siano c_1, \dots, c_5 i figli di w e siano k_1, \dots, k_4 le chiavi memorizzate in w . Per eliminare la condizione di overflow nel nodo w , eseguiamo un'operazione di *suddizione* o *scissione (split)* in w , come si può vedere nella Figura 11.24, in questo modo:

- Sostituiamo w con due nodi, w' e w'' , dove
 - w' è un 3-nodo che memorizza le chiavi k_1 e k_2 e ha i figli c_1, c_2 e c_3 .
 - w'' è un 2-nodo che memorizza la chiave k_4 e ha i figli c_4 e c_5 .
- Se w è la radice di T , creiamo un nuovo nodo radice, u , altrimenti sia u il genitore di w .
- Inseriamo la chiave k_3 in u e facciamo in modo che w' e w'' siano figli di u : se w era il figlio i -esimo di u , allora w' e w'' diventano, rispettivamente, i figli i -esimo e $(i + 1)$ -esimo di u .

Come conseguenza di un'operazione di suddizione del nodo w , può verificarsi una nuova situazione di overflow nel genitore u di w . Se questo accade, si innesta una suddizione nel nodo u (come si può vedere nella Figura 11.25). Quindi, una singola operazione di suddizione o elimina l'overflow o lo propaga al genitore del nodo in esame. La Figura 11.26 riporta una sequenza di inserimenti in un albero (2, 4).

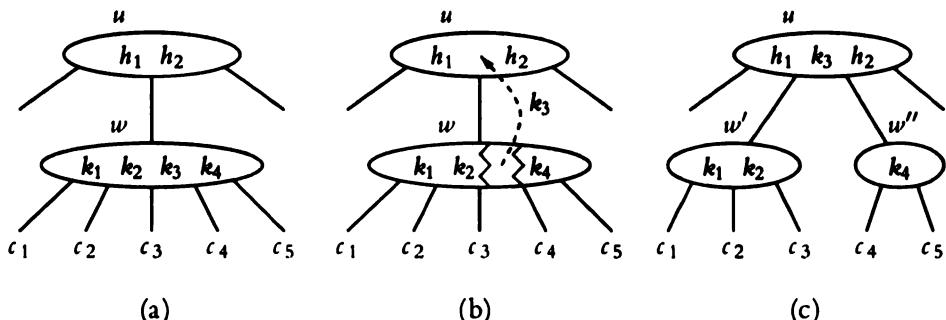


Figura 11.24: La suddizione di un nodo: (a) condizione di overflow in un 5-nodo, w ; (b) la terza chiave di w viene inserita nel genitore u di w ; (c) il nodo w viene sostituito da un 3-nodo, w' , e un 2-nodo, w'' .

Analisi dell'inserimento in un albero (2, 4)

Dato che il valore di d_{\max} è al massimo 4, la ricerca iniziale per l'inserimento della nuova chiave k impiega un tempo $O(1)$ per ciascun livello dell'albero, quindi un tempo complessivo $O(\log n)$, dal momento che l'altezza dell'albero, per la Proposizione 11.7, è $O(\log n)$.

Le modifiche che agiscono su un singolo nodo per l'inserimento di una nuova chiave e di un nuovo figlio possono essere realizzate in modo da richiedere un tempo d'esecuzione $O(1)$, così come una singola operazione di suddizione. Il numero di operazioni di suddizione che possono avvenire in cascata è limitato dall'altezza dell'albero, quindi la corrispondente fase della procedura di inserimento viene comunque eseguita in un tempo $O(\log n)$. Di conseguenza, il tempo totale richiesto per eseguire un inserimento in un albero (2, 4) è $O(\log n)$.

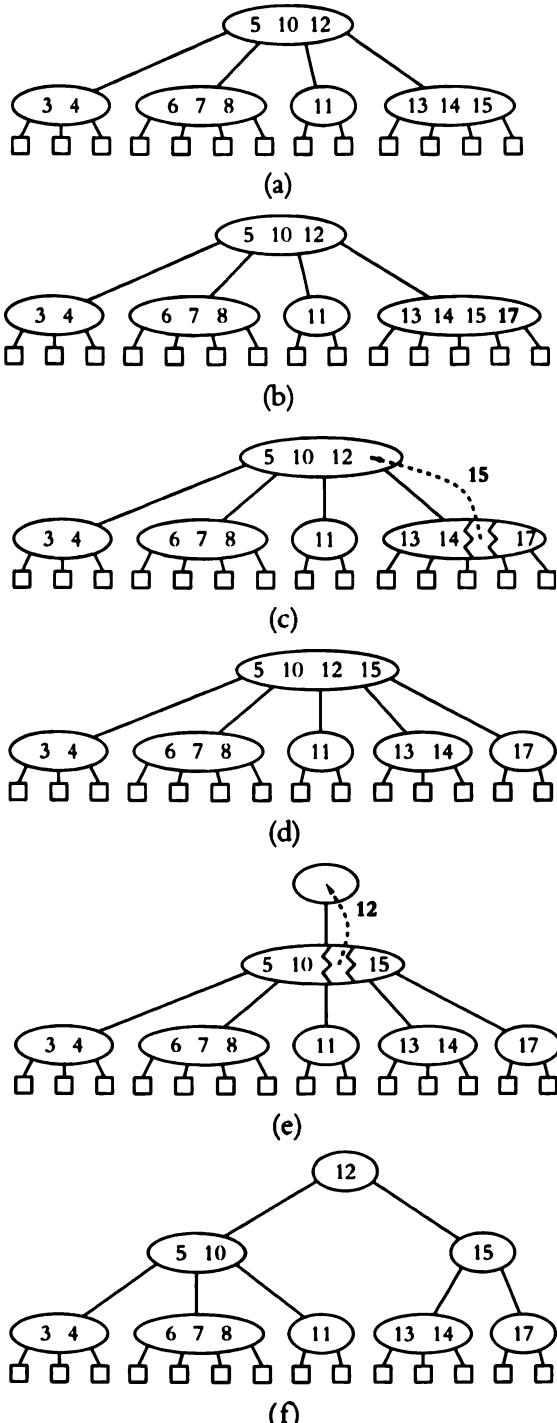


Figura 11.25: Un inserimento in un albero $(2, 4)$ che provoca una suddivisione a cascata:
 (a) prima dell'inserimento; (b) inserimento di 17, che provoca un overflow; (c) prima suddivisione;
 (d) dopo la suddivisione, si verifica un nuovo overflow; (e) seconda suddivisione, che crea
 un nuovo nodo radice; (f) situazione finale.

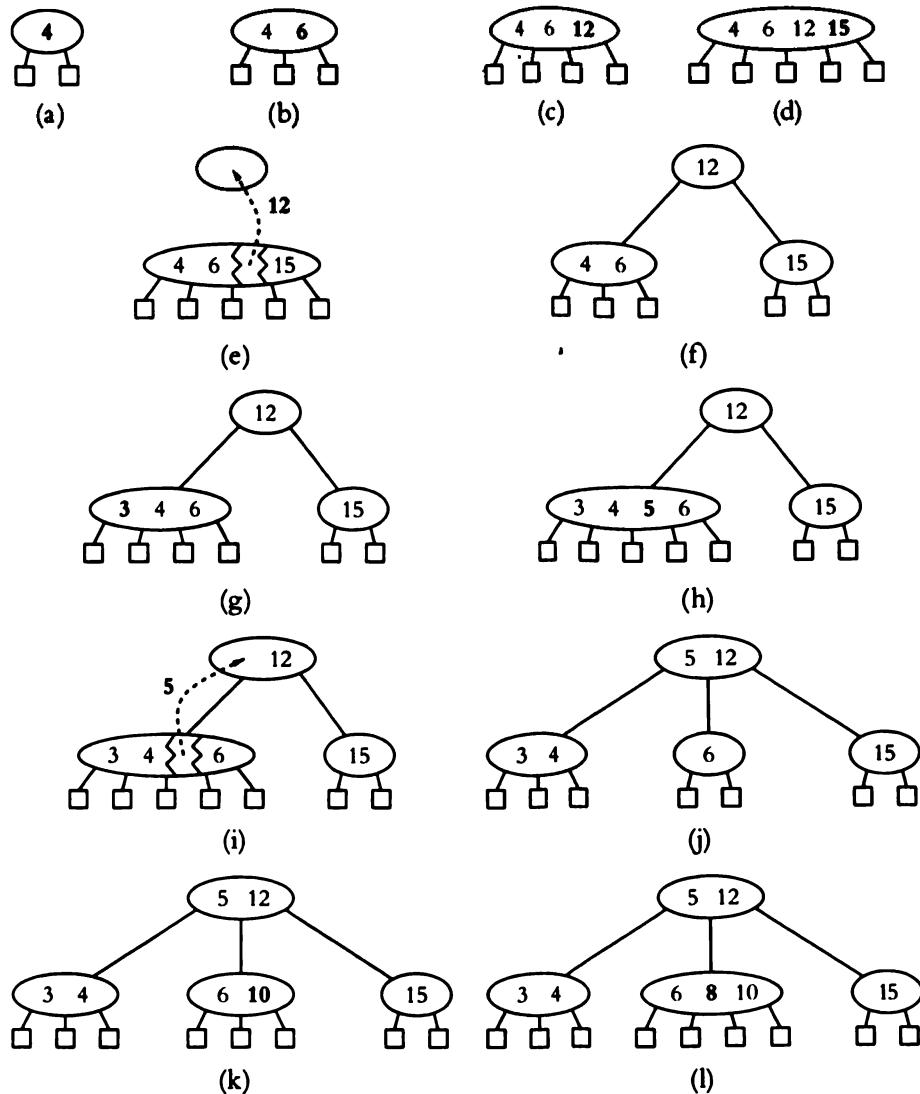


Figura 11.26: Una sequenza di inserimenti in un albero $(2, 4)$: (a) albero iniziale con una sola voce; (b) inserimento di 6; (c) inserimento di 12; (d) inserimento di 15, che provoca una situazione di overflow; (e) suddivisione, che causa la creazione di un nuovo nodo radice; (f) dopo la suddivisione; (g) inserimento di 3; (h) inserimento di 5, che provoca una situazione di overflow; (i) suddivisione; (j) dopo la suddivisione; (k) inserimento di 10; (l) inserimento di 8.

Rimozione

Consideriamo ora la rimozione da un albero $(2, 4)$ T della voce avente chiave k . Iniziamo la procedura eseguendo una ricerca in T della voce avente chiave k . La rimozione di una voce da un albero $(2, 4)$ può sempre essere ricondotta al caso in cui la voce da rimuovere è memorizzata in un nodo, w , i cui figli sono nodi esterni. Supponiamo, ad esempio, che la voce con chiave k che vogliamo rimuovere sia memorizzata nella i -esima voce (k_i, v_i) del nodo z che ha figli interni. In tal caso, scambiamo la voce (k_i, v_i) con un'opportuna voce che si trovi nel nodo w avente soltanto figli esterni (si veda la Figura 11.27d):

1. Cerchiamo il nodo interno w più a destra nel sottoalbero avente radice nel figlio i -esimo di z , osservando che i figli del nodo w sono tutti nodi esterni.
2. Scambiamo la voce (k_i, v_i) di z con l'ultima voce di w .

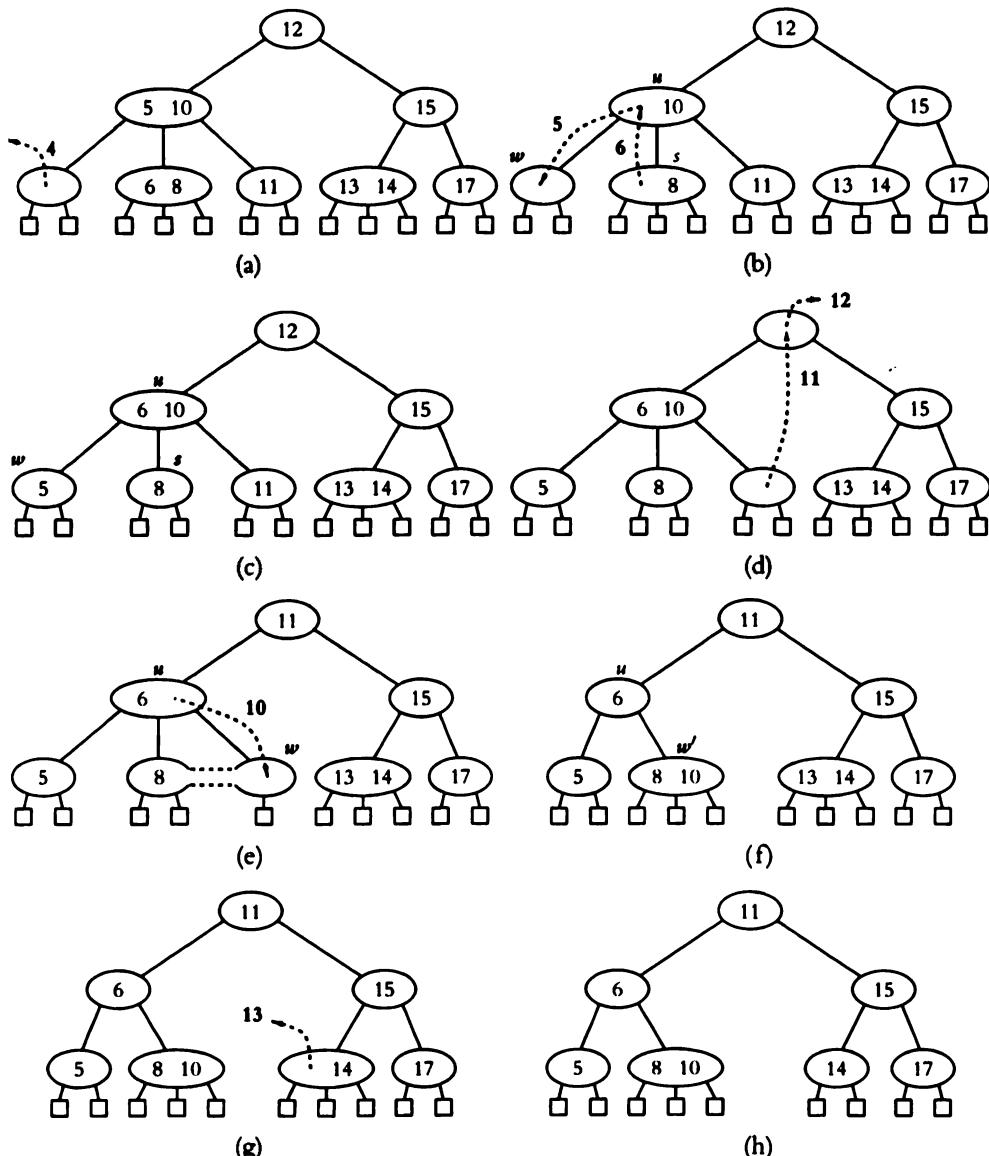


Figura 11.27: Una sequenza di rimozioni in un albero $(2, 4)$: (a) rimozione di 4, che provoca una situazione di underflow; (b) un'operazione di trasferimento; (c) dopo l'operazione di trasferimento; (d) rimozione di 12, che provoca una situazione di underflow; (e) un'operazione di fusione; (f) dopo l'operazione di fusione; (g) rimozione di 13; (h) dopo la rimozione di 13.

Una volta che siamo sicuri che la voce da eliminare sia memorizzata in un nodo w avente soltanto figli esterni (perché era già in w oppure ci finisce dopo uno scambio), eliminiamo semplicemente la voce da w , rimuovendo anche il nodo esterno che ricopre il ruolo di figlio i -esimo di w .

La rimozione di una voce (e di un figlio) da un nodo w seguendo la procedura appena delineata rispetta il vincolo sulla profondità, perché eliminiamo sempre un figlio esterno da un nodo w che ha soltanto figli esterni. Tuttavia, nel rimuovere tale nodo esterno, possiamo violare il vincolo sulla dimensione di w . Infatti, se prima della rimozione w era un 2-nodo, dopo la rimozione diventa un 1-nodo senza alcuna voce memorizzata al proprio interno (come si può vedere nelle Figure 11.27a e d), cosa che non è consentita in un albero (2, 4). Questo tipo di violazione del vincolo sulla dimensione è chiamato *underflow* (*sotto-dimensionamento*) del nodo w . Per porre rimedio a una situazione di underflow, verifichiamo se un fratello di w a esso adiacente sia un 3-nodo o un 4-nodo: se troviamo un tale fratello, s , eseguiamo un'operazione di *trasferimento*, assegnando un figlio di s a w , una chiave di s al genitore u di w e di s , e una chiave di u a w (come si può vedere nelle Figure 11.27b e c). Se w ha un solo fratello o se entrambi i fratelli di w adiacenti a esso sono 2-nodi, possiamo invece eseguire un'operazione di *fusione*, che fonde insieme w e un suo fratello, creando un nuovo nodo w' e spostando una chiave dal genitore u di w a w' (come si può vedere nelle Figure 11.27e e f).

Un'operazione di fusione che coinvolga il nodo w può provocare il verificarsi di una nuova condizione di underflow nel genitore u di w , che, a sua volta, può richiedere una fusione in u (si veda la Figura 11.28). Quindi, il numero di operazioni di fusione è limitato dall'altezza dell'albero, che è $O(\log n)$ per la Proposizione 11.7. Se una condizione di underflow si propaga fino alla radice dell'albero, questa viene semplicemente eliminata (come accade nelle Figure 11.28c e d).

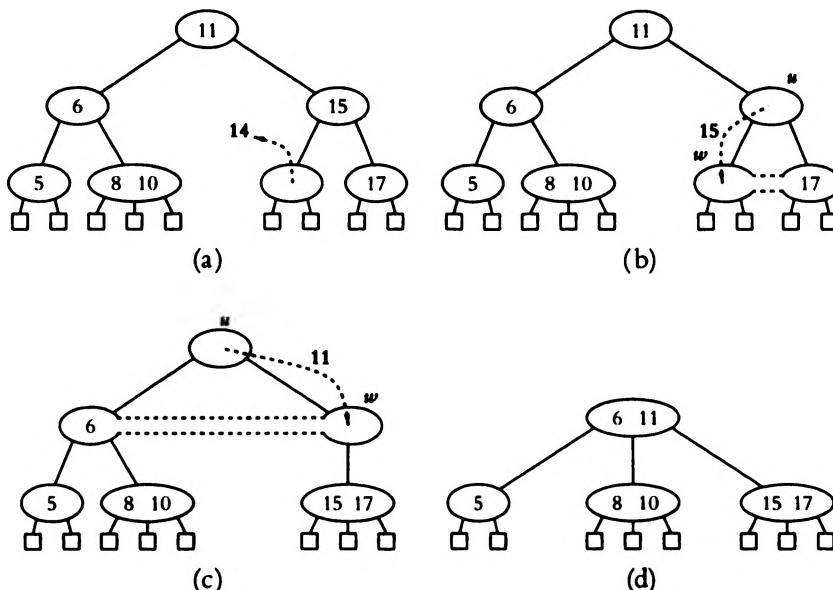


Figura 11.28: Propagazione di una sequenza di fusioni in un albero (2, 4): (a) rimozione di 14, che provoca una condizione di underflow; (b) fusione, che provoca un'altra condizione di underflow; (c) seconda operazione di fusione, che provoca la rimozione della radice; (d) albero finale.

Prestazioni degli alberi (2, 4)

Le prestazioni asintotiche dell'ADT "mappa ordinata" realizzata con un albero (2, 4) sono identiche a quelle di una mappa ordinata realizzata con un albero AVL (riassunte nella Tabella 11.2), con limite logaritmico garantito per la maggior parte delle operazioni. L'analisi della complessità temporale delle operazioni in un albero (2, 4) contenente n coppie chiave-valore si basa sulle seguenti osservazioni.

- L'altezza di un albero (2, 4) contenente n voci è $O(\log n)$, per la Proposizione 11.7.
- Un'operazione di suddivisione, di trasferimento o di fusione richiede un tempo $O(1)$.
- Un'operazione di ricerca, inserimento o rimozione coinvolge $O(\log n)$ nodi.

Quindi, l'albero (2, 4) consente l'esecuzione veloce delle operazioni fondamentali di una mappa: ricerca, inserimento e rimozione. Gli alberi (2, 4) hanno anche una relazione interessante con la struttura dati che presenteremo nel prossimo paragrafo.

11.6 Alberi rosso-nero

Anche se gli alberi AVL e gli alberi (2, 4) hanno diverse proprietà interessanti, presentano qualche svantaggio. Ad esempio, gli alberi AVL possono avere bisogno di molte operazioni di ristrutturazione (rotazioni) dopo un'operazione di rimozione, e gli alberi (2, 4) possono richiedere l'esecuzione di molte operazioni di suddivisione o di fusione dopo un inserimento o, rispettivamente, una rimozione. La struttura dati che presentiamo in questo paragrafo, l'albero rosso-nero, non ha questi svantaggi: per rimanere bilanciato, richiede un numero $O(1)$ di modifiche strutturali dopo un'operazione di inserimento o rimozione.

Formalmente, un *albero rosso-nero* (*red-black tree*) è un albero di ricerca binario (come definito nel Paragrafo 11.1) con nodi colorati di rosso o di nero in modo da soddisfare le seguenti proprietà:

Proprietà della radice: La radice è nera.

Proprietà delle foglie: I nodi esterni sono neri.

Proprietà dei rossi: I figli di un nodo rosso sono neri.

Vincolo sulla profondità: Tutti i nodi esterni hanno la stessa *profondità nera*, definita come il numero di antenati *propri* che sono neri.

La Figura 11.29 mostra un esempio di albero rosso-nero.

Possiamo rendere più intuitiva la definizione di albero rosso-nero osservando un'interessante correlazione tra alberi rosso-neri e alberi (2, 4). Dato un albero rosso-nero, possiamo costruire l'albero (2, 4) corrispondente fondendo ogni nodo rosso w nel suo genitore, memorizzando le voci di w nel suo genitore e facendo in modo che i figli di w diventino ordinatamente figli del suo genitore. Ad esempio, l'albero rosso-nero della Figura 11.29 corrisponde all'albero (2, 4) della Figura 11.23, come si può capire meglio osservando la

Figura 11.30: Il vincolo sulla profondità degli alberi rosso-nero corrisponde al vincolo sulla profondità degli alberi $(2, 4)$, perché uno è un solo nodo nero dell'albero rosso-nero contribuisce a ciascun nodo del corrispondente albero $(2, 4)$.

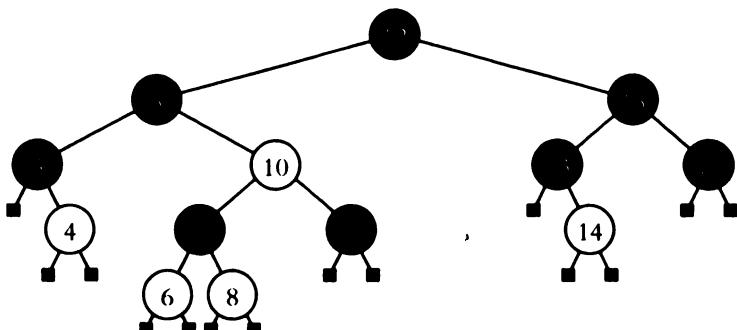


Figura 11.29: Un esempio di albero rosso-nero (i nodi "rossi" sono evidentemente rappresentati in bianco). Tutte le foglie di quest'albero hanno "profondità nera" uguale a 3.

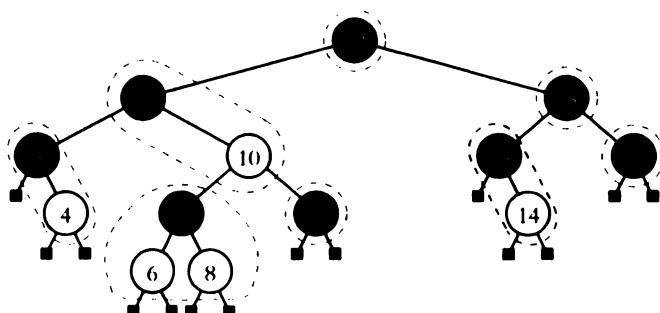


Figura 11.30: Corrispondenza tra l'albero rosso-nero della Figura 11.29 e l'albero $(2, 4)$ della Figura 11.23, sulla base dei raggruppamenti tra nodi rossi e il loro genitore nero che abbiamo evidenziato.

Al contrario, possiamo trasformare un albero $(2, 4)$ nel corrispondente albero rosso-nero colorando di nero tutti i nodi e, poi, eseguendo le seguenti trasformazioni su ciascun nodo w , come visibile nella Figura 11.31:

- Se w è un 2-nodo, i suoi figli rimangono neri.
- Se w è un 3-nodo, creiamo un nuovo nodo rosso, y , e trasferiamo a y i due ultimi figli di w (che rimangono neri), poi facciamo in modo che i due figli di w siano l'attuale primo figlio di w , seguito da y .
- Se w è un 4-nodo, creiamo due nuovi nodi rossi, y e z , poi trasferiamo a y i primi due figli di w (che rimangono neri), trasferiamo a z gli ultimi due figli di w (che rimangono neri) e, infine, facciamo in modo che i due figli di w siano y e, nell'ordine, z .

Si noti che, in questa costruzione, un nodo rosso ha sempre un genitore nero.

Proposizione 11.8: L'altezza di un albero rosso-nero che memorizza n voci è $O(\log n)$.

Dimostrazione: Sia T un albero rosso-nero che memorizza n voci e sia h l'altezza di T . Dimostreremo questa proposizione dimostrando che:

$$\log(n + 1) \leq h \leq 2\log(n + 1).$$

Sia d la "profondità nera" comune a tutti i nodi esterni di T . Sia T' l'albero (2, 4) associato a T e sia h' l'altezza di T' . Per la corrispondenza esistente tra alberi rosso-nero e alberi (2, 4), sappiamo che $h' = d$, quindi, per la Proposizione 11.7, $d = h' \leq \log(n + 1)$. Per la proprietà dei rossi, $h \leq 2d$. Quindi, otteniamo che $h \leq 2 \log(n + 1)$. L'altra diseguaglianza, $\log(n + 1) \leq h$, discende dalla Proposizione 8.7 e dal fatto che T ha n nodi interni. ■

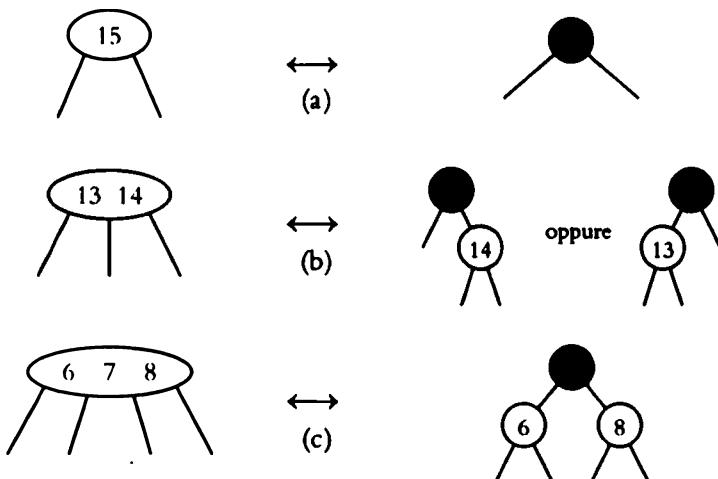


Figura 11.31: Corrispondenza tra i nodi di un albero (2, 4) e quelli di un albero rosso-nero:
(a) 2-nodo; (b) 3-nodo; (c) 4-nodo.

11.6.1 Operazioni in un albero rosso-nero

L'algoritmo di ricerca in un albero rosso-nero T è identico a quello che agisce su un albero di ricerca binario standard, descritto nel Paragrafo 11.1. Quindi, la ricerca in un albero rosso-nero richiede un tempo proporzionale all'altezza dell'albero, che, per la Proposizione 11.8, è $O(\log n)$.

La corrispondenza esistente tra alberi (2, 4) e alberi rosso-nero ci fornisce alcuni suggerimenti importanti, che useremo nella nostra discussione in merito a come si possano effettuare modifiche in alberi rosso-nero: in effetti, senza queste similitudini, gli algoritmi di modifica degli alberi rosso-nero possono sembrare veramente complessi e misteriosi. Le operazioni di suddivisione e fusione degli alberi (2, 4) saranno sostituite efficacemente da azioni di cambiamento di colore di nodi adiacenti nell'albero rosso-nero. Per cambiare l'orientazione di un 3-nodo, scambiandola tra le due forme della Figura 11.31b, verrà usata una rotazione all'interno dell'albero rosso-nero.

Inserimento

Consideriamo l'inserimento di una coppia chiave-valore (k, v) in un albero rosso-nero T . L'algoritmo procede inizialmente come in un albero di ricerca binario (Paragrafo 11.1.2): cerchiamo la chiave k in T e se raggiungiamo un nodo esterno lo sostituiamo con un nuovo nodo interno x che memorizza la nuova voce e che abbia due figli esterni. Se stiamo inserendo la prima voce in T e, quindi, x è la radice, lo coloriamo di nero; in tutti gli altri casi, coloriamo x di rosso. Questa azione corrisponde all'inserimento di (k, v) in un nodo del livello interno più basso dell'albero $(2, 4)$ equivalente, T' . L'inserimento preserva la proprietà della radice e il vincolo sulla profondità di T , ma può violare la proprietà dei rossi. Infatti, se x non è la radice di T e il suo genitore y è rosso, ci ritroviamo con un figlio e il suo genitore (rispettivamente, x e y) che sono entrambi rossi. Osserviamo che, per la proprietà della radice, y non può essere la radice di T , mentre per la proprietà dei rossi (che prima dell'inserimento era soddisfatta), il genitore z di y deve essere nero. Dato che x e il suo genitore sono rossi, ma z , il genitore del genitore di x , è nero, questa violazione della proprietà dei rossi è chiamata *doppio rosso* nel nodo x . Per sistemare un doppio rosso, dobbiamo prendere in esame due casi.

Caso 1: Il fratello s di y è un nodo nero. In questo caso, come si può vedere nella Figura 11.32, il doppio rosso è dovuto all'inserimento del nuovo nodo in una posizione corrispondente a un 3-nodo nell'albero $(2, 4)$, T' , creando a tutti gli effetti un 4-nodo non corretto. Questa configurazione ha un nodo rosso, y , che è genitore di un altro nodo rosso, x : vogliamo, invece, che i due nodi rossi diventino fratelli. Per risolvere questo problema, eseguiamo una *ristrutturazione di una terna di nodi* in T . Tale ristrutturazione, introdotta nel Paragrafo 11.2, viene eseguita dall'operazione `restructure(x)`, che è composta dalle seguenti fasi (riportate, di nuovo, nella Figura 11.32):

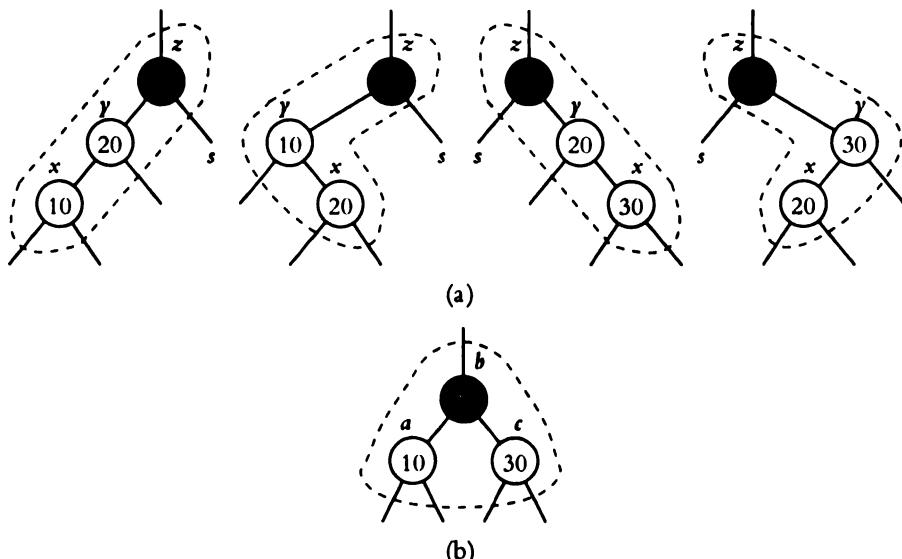


Figura 11.32: Ristrutturazione di un albero rosso-nero per eliminare una configurazione doppio rosso: (a) le quattro configurazioni possibili per i nodi x , y e z prima della ristrutturazione; (b) dopo la ristrutturazione.

- Prendi il nodo x , il suo genitore y e il genitore di y, z , etichettandoli temporaneamente come a, b e c , in ordine da sinistra a destra, in modo che a, b e c vengano visitati in tale ordine dall'attraversamento simmetrico dell'albero.
- Sostituisce z con il nodo b e fa in modo che a e c siano i figli di b , mantenendo inalterata la disposizione relativa sulla base dell'attraversamento in ordine simmetrico.

Dopo aver eseguito l'operazione `restructure(x)`, coloriamo b di nero e a e c di rosso, eliminando, così, il problema del doppio rosso. Osserviamo che la porzione di qualsiasi percorso che attraversi la parte di albero ristrutturata contiene uno e un solo nodo nero, tanto prima della ristrutturazione quanto dopo di essa, per cui la profondità nera dell'albero non viene modificata.

Caso 2: Il fratello s di y è un nodo rosso. In questo caso, come si può vedere nella Figura 11.33, il doppio rosso è dovuto a un overflow nel corrispondente albero (2,4), T' . Per risolvere questo problema, eseguiamo l'equivalente di un'operazione di suddivisione (*split*), che è un *cambiamento di colorazione* (*recoloring*): coloriamo y e s di nero e il loro genitore z di rosso (a meno che z non sia la radice, nel qual caso rimane nero). Si noti che, se z non è la radice, la porzione di qualsiasi percorso che attraversi la parte di albero coinvolta dall'operazione contiene uno e un solo nodo nero, tanto prima della colorazione quanto dopo di essa, per cui la profondità nera dell'albero non viene modificata, tranne quando z è la radice, nel qual caso aumenta di uno.

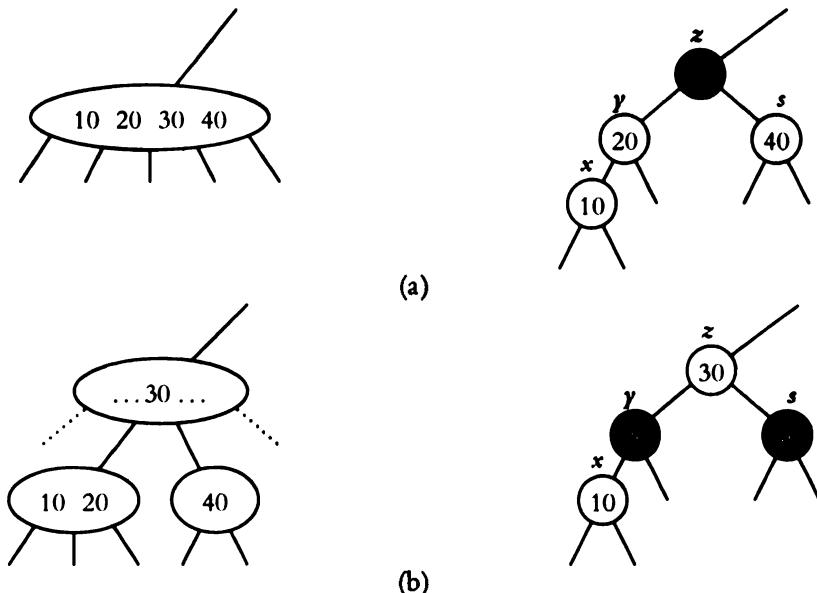


Figura 11.33: Cambiamento di colorazione per porre rimedio a un problema di doppio rosso:
(a) porzione di albero rosso-nero prima del cambiamento di colorazione e 5-nodo nell'albero (2,4) corrispondente, prima della suddivisione; (b) dopo il cambiamento di colorazione e dopo la suddivisione.

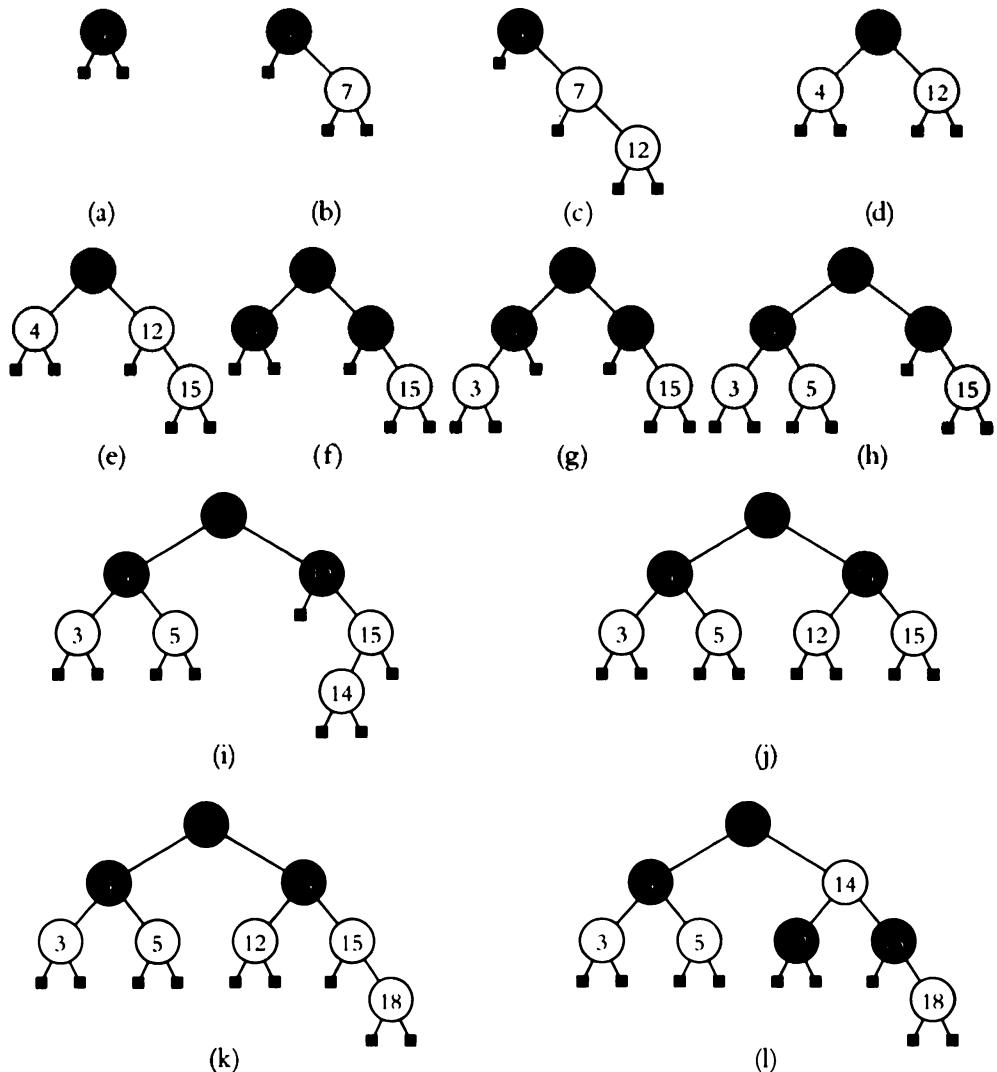


Figura 11.34: Una sequenza di inserimenti in un albero rosso-nero (che prosegue nella Figura 11.35): (a) albero iniziale; (b) inserimento di 7; (c) inserimento di 12, che provoca un doppio rosso; (d) dopo la ristrutturazione; (e) inserimento di 15, che provoca un doppio rosso; (f) dopo il cambiamento di colorazione (la radice rimane nera); (g) inserimento di 3; (h) inserimento di 5; (i) inserimento di 14, che provoca un doppio rosso; (j) dopo la ristrutturazione; (k) inserimento di 18, che provoca un doppio rosso; (l) dopo il cambiamento di colorazione.

È, però, possibile che il problema del doppio rosso ricompaia dopo un tale cambiamento di colorazione, anche se più in alto nell'albero T , perché z potrebbe avere un genitore rosso. Se il problema del doppio rosso ricompare in z , ripetiamo la distinzione tra i due casi a partire da z . Quindi, un cambiamento di colorazione elimina il problema del

doppio rosso nel nodo x , oppure lo propaga a z , genitore del genitore di x . Si continua a risalire in T effettuando cambiamenti di colorazione finché non si risolve in modo definitivo il problema del doppio rosso (con un ultimo cambiamento di colorazione o una ristrutturazione di una terna di nodi). Quindi, il numero di cambiamenti di colorazione provocato da un inserimento non è maggiore della metà dell'altezza dell'albero T , che, per la Proposizione 11.8, è $O(\log n)$.

Come ulteriore esempio, le Figure 11.34 e 11.35 mostrano una sequenza di operazioni di inserimento in un albero rosso-nero.

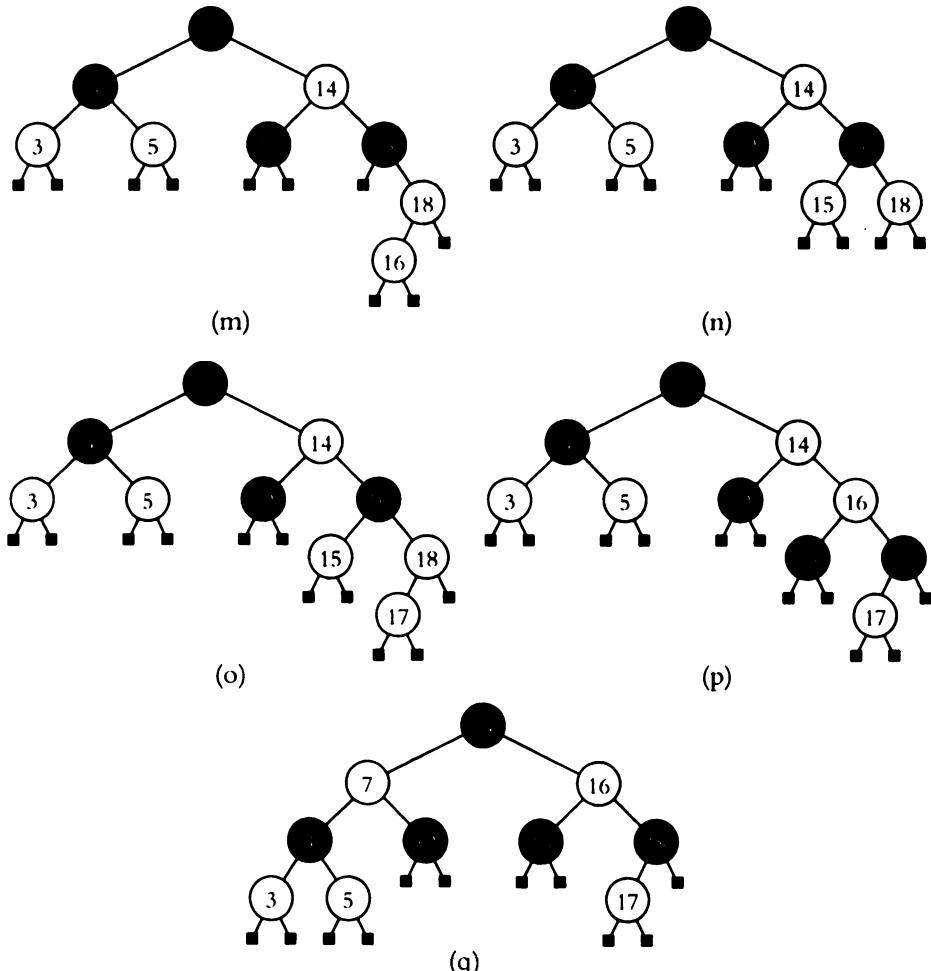


Figura 11.35: Una sequenza di inserimenti in un albero rosso-nero (che continua dalla Figura 11.34): (m) inserimento di 16, che provoca un doppio rosso; (n) dopo la ristrutturazione; (o) inserimento di 17, che provoca un doppio rosso; (p) dopo il cambiamento di colorazione c'è di nuovo un doppio rosso, da gestire mediante ristrutturazione; (q) dopo la ristrutturazione.

Rimozione

La procedura di rimozione della voce avente chiave k da un albero rosso-nero T inizia come quella vista nel Paragrafo 11.1.2 per un albero di ricerca binario standard. Strutturalmente, la procedura consiste nella rimozione di un nodo interno (quello che conteneva effettivamente la chiave k , oppure il suo predecessore nella disposizione descritta dall'attraversamento in ordine simmetrico) insieme a uno dei suoi figli che sia esterno, il tutto seguito dalla promozione dell'altro figlio.

Se il nodo interno rimosso era rosso, questa modifica strutturale non influenza la profondità nera di nessun percorso nell'albero, né introduce una violazione della proprietà dei rossi, per cui l'albero risultante è ancora un albero rosso-nero valido. Nel corrispondente albero $(2, 4)$, T' , questo caso si risolve con la diminuzione di dimensione di un 4-nodo o di un 3-nodo. Se il nodo interno rimosso era nero, doveva avere altezza nera uguale a 1 e, quindi, entrambi i suoi figli erano esterni oppure uno dei suoi figli era un nodo interno rosso con due figli esterni. Nel secondo caso, il nodo rimosso rappresenta la parte nera di un corrispondente 3-nodo e le proprietà dell'albero rosso-nero vengono ripristinate con un cambiamento di colorazione del figlio promosso, che diventa nero.

Il caso più complesso si ha quando il nodo rimosso era nero e aveva due figli esterni. Nel corrispondente albero $(2, 4)$, questo indica la rimozione di una voce da un 2-nodo. Senza ripristinare il bilanciamento, una tale modifica provoca un valore troppo basso per la profondità nera della posizione esterna p che contiene il figlio promosso del nodo interno eliminato. Per rispettare il vincolo sulla profondità, assegniamo temporaneamente alla foglia promossa un colore fittizio, denominato *doppio nero* (*double black*). Un doppio nero in T segnala un underflow nel corrispondente albero $(2, 4)$, T' . Per eliminare il problema del doppio nero in una posizione p qualsiasi, considereremo tre casi.

Caso 1: *Il fratello y di p è nero e ha un figlio rosso, x .* La situazione è rappresentata nella Figura 11.36. Eseguiamo una *ristrutturazione di una terna di nodi*, così come descritto nel Paragrafo 11.2. L'operazione *restructure(x)* prende il nodo x , il suo genitore y e il genitore del genitore, z , e li etichetta temporaneamente da sinistra a destra come a , b e c ; poi, sostituisce z con b , facendo in modo che diventi il genitore degli altri due. Coloriamo a e c di nero e assegniamo a b il colore che aveva precedentemente z .

Osserviamo che il percorso verso p nell'albero risultante contiene un nodo nero in più rispetto alla situazione iniziale, mentre il numero di nodi neri che si trovano sui percorsi che vanno agli altri tre sottoalberi illustrati nella Figura 11.36 rimane immutato. Quindi, riportiamo p al (normale) colore nero e il problema del doppio nero è stato eliminato. La soluzione di questo caso corrisponde, nell'albero $(2, 4)$ T' , all'operazione di trasferimento tra due figli del nodo z . Il fatto che y abbia un figlio rosso assicura che y rappresenti un 3-nodo o un 4-nodo. All'atto pratico, la voce precedentemente memorizzata in z viene retrocessa per diventare un nuovo 2-nodo, per risolvere la mancanza, mentre si promuove una delle voci memorizzate in y o suo figlio, per prendere il posto della voce che prima si trovava in z .

Caso 2: *Il fratello y di p è nero come i suoi due figli.* Eseguiamo un *cambiamento di colorazione*, iniziando con il cambiamento del colore di p , che passa da "doppio nero" a nero, e di y , che passa da nero a rosso. Queste modifiche non creano violazioni alla proprietà dei rossi, perché entrambi i figli di y sono neri. Per reagire al decremento della profondità

nera dei percorsi che contengono y o p , consideriamo il genitore comune di p e y , che chiamiamo z . Se z è rosso, lo coloriamo di nero e il problema è risolto (come si può vedere nella Figura 11.37a). Se, invece, z è nero, lo coloriamo in *doppio nero*, propagando così il problema verso l'alto nell'albero (come si può vedere nella Figura 11.37b). La soluzione di questo caso corrisponde, nell'albero (2, 4) T' , a un'operazione di fusione, perché y deve rappresentare un 2-nodo. Il caso in cui il problema si propaga verso l'alto corrisponde alla situazione in cui anche il genitore z rappresenta un 2-nodo.

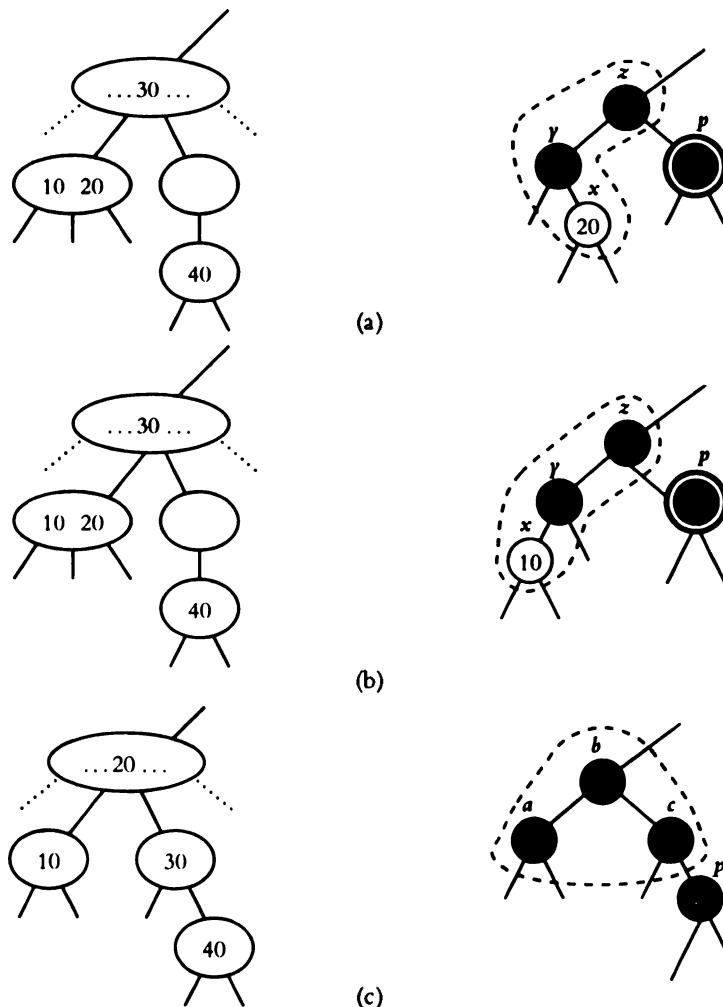


Figura 11.36: Ristrutturazione di un albero rosso-nero per porre rimedio al problema del doppio nero: (a, b) configurazioni prima della ristrutturazione, in cui p è un figlio destro e nell'albero (2, 4) corrispondente vengono mostrati i nodi associati, prima del trasferimento (sono possibili due altre configurazioni, simmetriche, relative al caso in cui p sia un figlio sinistro); (c) configurazione dopo la ristrutturazione e, nell'albero (2, 4) corrispondente, i nodi associati dopo il trasferimento. Il colore grigio del nodo z in (a) e in (b) e del nodo b in (c) segnala che questo nodo può essere tanto rosso quanto nero.

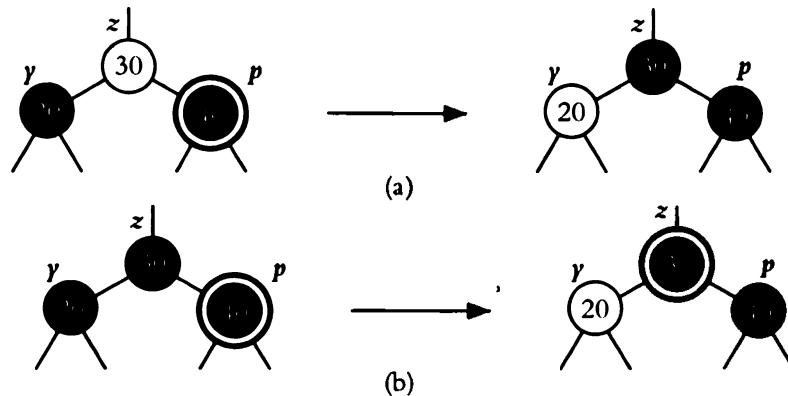


Figura 11.37: Un'operazione di cambiamento di colorazione, che non incide sulla profondità nera dei percorsi: (a) quando z era inizialmente rosso, il cambiamento di colorazione risolve il problema del doppio nero, terminando la procedura; (b) quando z era inizialmente nero, diventa "doppio nero" e innesta una procedura di soluzione a cascata.

Caso 3: Il fratello y di p è rosso. Indichiamo con z il genitore comune di y e p e osserviamo che z deve essere nero, perché y è rosso. La combinazione di y e z rappresenta un 3-nodo nel corrispondente albero $(2, 4) T'$. In questo caso, eseguiamo una rotazione relativa a y e z , poi cambiamo il colore di y , che diventa nero, e di z , che diventa rosso. Tutto questo equivale a un cambiamento dell'orientazione di un 3-nodo nel corrispondente albero $(2, 4) T'$. Ora riprendiamo in esame il problema del doppio nero in p . Dopo le modifiche fatte, il fratello di p è nero e si applica il Caso 1 o il Caso 2. Inoltre, la prossima modifica sarà l'ultima, perché il Caso 1 porta sempre a termine la procedura e il Caso 2 sarà anch'esso conclusivo, dato che ora il genitore di p è rosso.

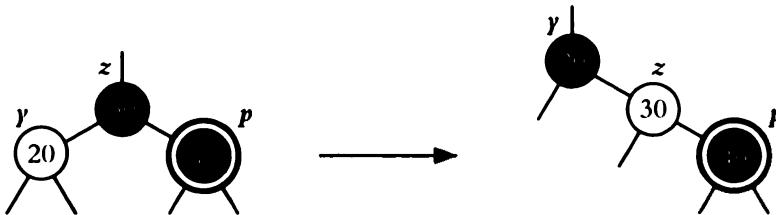


Figura 11.38: Una rotazione e un cambiamento di colorazione del nodo rosso y e del nodo nero z in presenza di un problema di doppio nero (esiste anche una configurazione simmetrica a questa). Tutto questo equivale a un cambiamento dell'orientazione del corrispondente 3-nodo nell'albero $(2, 4)$ e questa operazione non ha effetti sulla profondità nera di alcun percorso che attraversi questa porzione dell'albero. Dopo l'operazione, però, occorre applicare uno degli altri casi di soluzione del problema doppio nero, perché il fratello di p sarà diventato nero.

Nella Figura 11.39 mostriamo una sequenza di rimozioni in un albero rosso-nero. Nei punti (c) e (d) abbiamo illustrato una ristrutturazione corrispondente al Caso 1, mentre nei punti (f) e (g) è stato riportato un cambiamento di colorazione relativo al Caso 2. Infine, i punti (i) e (j) mostrano un esempio della rotazione del Caso 3, che si conclude con un cambiamento di colorazione conseguente all'applicazione del Caso 2, nel punto (k).

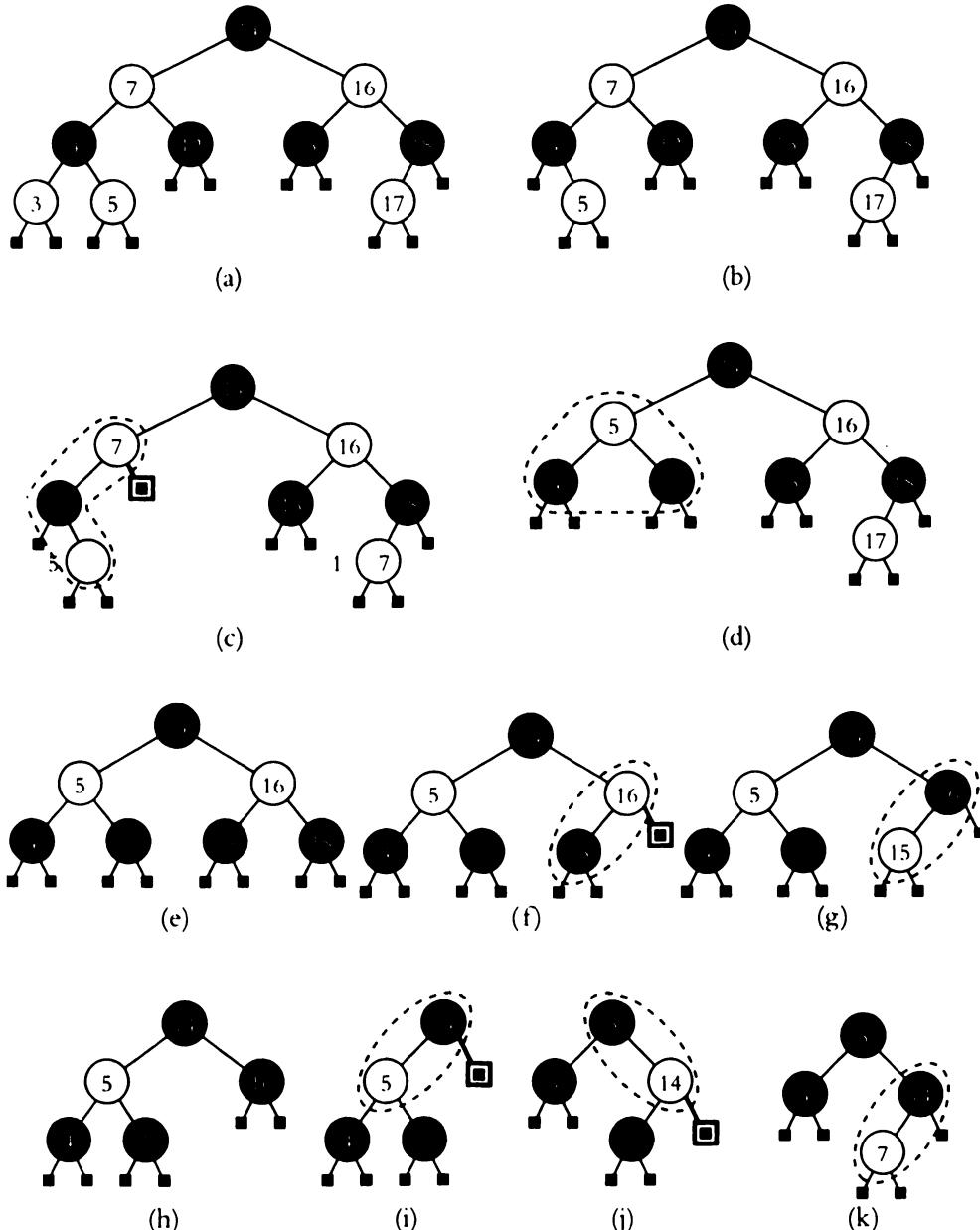


Figura 11.39: Una sequenza di rimozioni in un albero rosso-nero: (a) albero iniziale; (b) rimozione di 3; (c) rimozione di 12, che provoca un deficit di nero a destra di 7 (risolto con una ristrutturazione); (d) dopo la ristrutturazione; (e) rimozione di 17; (f) rimozione di 18, che provoca un deficit di nero a destra di 16 (risolto con un cambiamento di colorazione); (g) dopo il cambiamento di colorazione; (h) rimozione di 15; (i) rimozione di 16, che provoca un deficit di nero a destra di 14 (inizialmente gestito con una rotazione); (j) dopo la rotazione, il deficit di nero deve essere gestito da un cambiamento di colorazione; (k) dopo tale ultimo cambiamento di colorazione.

Prestazioni degli alberi rosso-nero

Le prestazioni asintotiche degli alberi rosso-nero come implementazione del tipo di dato astratto “mappa ordinata” sono identiche a quelle degli alberi AVL e degli alberi (2, 4), con garanzia di un tempo d’esecuzione superiormente limitato da una funzione logaritmica per le operazioni principali (le prestazioni degli alberi AVL, identiche, sono state riassunte nella Tabella 11.2). Il vantaggio principale degli alberi rosso-nero risiede nel fatto che un’operazione di inserimento o di rimozione richiede soltanto *un numero costante di operazioni di ristrutturazione*, diversamente da quanto avviene negli alberi AVL e negli alberi (2, 4), dato che entrambe quelle strutture necessitano, nel caso peggiore, di un numero di modifiche strutturali logaritmico per una singola operazione di aggiornamento eseguita sulla mappa. Invece, un’operazione di inserimento o rimozione in un albero rosso-nero richiede un tempo logaritmico per effettuare la ricerca iniziale e può richiedere un numero logaritmico di operazioni di cambiamento di colorazione, che risalgono l’albero. Queste osservazioni vengono ora formalizzate nelle proposizioni che seguono.

Proposizione 11.9: *L’inserimento di una voce in un albero rosso-nero che contenga n voci può essere fatta in un tempo $O(\log n)$ e richiede al massimo una operazione di ristrutturazione di una terna di nodi e un numero $O(\log n)$ di cambiamenti di colorazione.*

Proposizione 11.10: *La rimozione di una voce da un albero rosso-nero che contenga n voci può essere fatta in un tempo $O(\log n)$ e richiede al massimo due operazioni di ristrutturazione di una terna di nodi e un numero $O(\log n)$ di cambiamenti di colorazione.*

Le dimostrazioni di queste proposizioni sono rimandate all’Esercizio R-11.26 e all’Esercizio R-11.27.

11.6.2 Implementazione in Java

In questo paragrafo forniremo un’implementazione della classe `RBTreeMap`, (*RB* sta per red-black, cioè rosso-nero), derivata dalla classe `TreeMap` e basata sull’infrastruttura di bilanciamento descritta nel Paragrafo 11.2.1, dove era previsto che ogni nodo avesse un’informazione ausiliaria sotto forma di numero intero, utilizzabile per gestire il bilanciamento. In un albero rosso-nero, usiamo quel numero intero per rappresentare il colore, scegliendo il valore 0 (che è il valore predefinito per quel tipo di variabile) per il colore nero e il valore 1 per il colore rosso: con questa convenzione, ogni foglia creata nell’albero sarà nera.

La nostra implementazione inizia nel Codice 11.14, con i costruttori di una mappa vuota e una serie di metodi ausiliari per la gestione del campo che rappresenta le informazioni di colorazione. Quel codice prosegue con il supporto per il ripristino del bilanciamento dell’albero dopo l’esecuzione di un’operazione di inserimento. Dopo che una voce è stata inserita nell’albero dall’algoritmo standard degli alberi di ricerca, sarà memorizzata in un nodo che, prima dell’inserimento, era esterno e che è stato convertito in nodo interno avente due figli esterni. Viene, poi, invocato il metodo `rebalanceInsert`, strategia che ci dà l’opportunità di apportare modifiche all’albero. Tranne nel caso speciale in cui il nuovo elemento si trova nella radice, modifichiamo il colore del nodo che contiene il nuovo elemento, facendolo diventare rosso (era nero quando il nodo era un foglia), poi verifichiamo se sussista una violazione di tipo “doppio rosso”. Il metodo ausiliario `resolveRed`, che risolve

questo eventuale problema, segue da vicino l'analisi in più casi che abbiamo descritto nel Paragrafo 11.6.1., invocando se stesso ricorsivamente nel caso in cui la violazione si propaghi verso l'alto.

Il Codice 11.15, invece, gestisce la procedura di ripristino del bilanciamento dopo una rimozione, basandosi sull'analisi, sempre in più casi, che abbiamo descritto nello stesso Paragrafo 11.6.1. Se il nodo rimosso era rosso, non è richiesta nessuna ulteriore azione, ma, se il nodo rimosso era nero, dobbiamo ripristinare il vincolo sulla profondità. Una sfida ulteriore è rappresentata dal fatto che, nel momento in cui è stato invocato il metodo `rebalanceDelete`, il nodo da rimuovere era già stato rimosso dall'albero (il metodo viene, infatti, invocato usando come parametro il figlio promosso di quel nodo rimosso).

In particolare, indichiamo con p il figlio promosso del nodo eliminato. Se è stato eliminato un nodo nero con un figlio rosso, allora p sarà quel figlio rosso: risolviamo il problema colorando p di nero. Altrimenti, se p non è la radice, indichiamo con s il fratello del nodo eliminato (che, dopo quella eliminazione, sarà il fratello di p). Se il nodo eliminato era nero e aveva due figli neri, dobbiamo considerare p come un nodo *doppio nero*, da sistemare. Questo accade se, e solo se, il sottoalbero di suo fratello ha un nodo interno nero (perché il vincolo sulla profondità degli albero rosso-nero era rispettato prima della rimozione avvenuta). Quindi, verifichiamo se s è un nodo interno nero oppure un nodo interno rosso con un nodo interno come figlio (che dovrà essere nero per la proprietà dei rossi).

Siamo, quindi, in grado di individuare un problema "doppio nero" all'interno del metodo `rebalanceDelete` nel Codice 11.15 e risolviamo il problema usando il metodo ricorsivo `remedyDoubleBlack`.

Codice 11.14: La classe `RBTreeMap` (prosegue nel Codice 11.15).

```

1  /** Un'implementazione di mappa ordinata che usa un albero rosso-nero. */
2  public class RBTreeMap<K,V> extends TreeMap<K,V> {
3      /** Costruisce una mappa vuota usando l'ordinamento naturale tra le chiavi. */
4      public RBTreeMap() { super(); }
5      /** Costruisce una mappa vuota ordinando le chiavi con il comparatore dato. */
6      public RBTreeMap(Comparator<K> comp) { super(comp); }
7      // usiamo il campo ausiliario ereditato con 0=nero e 1=rosso
8      // (così le nuove foglie saranno nere)
9      private boolean isBlack(Position<Entry<K,V>> p) { return tree.getAux(p)==0; }
10     private boolean isRed(Position<Entry<K,V>> p) { return tree.getAux(p)==1; }
11     private void makeBlack(Position<Entry<K,V>> p) { tree.setAux(p, 0); }
12     private void makeRed(Position<Entry<K,V>> p) { tree.setAux(p, 1); }
13     private void setColor(Position<Entry<K,V>> p, Boolean toRed) {
14         tree.setAux(p, toRed ? 1 : 0);
15     }
16     /** Sovrascrive il metodo di TreeMap che viene invocato dopo un inserimento. */
17     protected void rebalanceInsert(Position<Entry<K,V>> p) {
18         if (!isRoot(p)) {
19             makeRed(p);    // il nuovo nodo interno viene inizialmente colorato di rosso
20             resolveRed(p); // ma questo può provocare un problema "doppio rosso"
21         }
22     }
23     /** Sistema potenziali problemi "doppio rosso" al di sopra del nodo rosso p. */
24     private void resolveRed(Position<Entry<K,V>> p) {
25         Position<Entry<K,V>> parent,uncle,middle,grand; // usati nell'analisi dei casi
26         parent = parent(p);
27         if (isRed(parent)) {                                // il problema del doppio rosso esiste

```

```

uncle = sibling(parent);    // è lo zio, cioè il fratello del genitore
if (isBlack(uncle)) {        // Caso 1: 4-nodo con forma anomala
    middle = restructure(p); // ristrutturazione di una terna di nodi
    makeBlack(middle);
    makeRed(left(middle));
    makeRed(right(middle));
} else {                     // Caso 2: 5-nodo in overflow
    makeBlack(parent);       // cambiamento di colorazione
    makeBlack(uncle);
    grand = parent(parent); // è il nonno, cioè il genitore del genitore
    if (!isRoot(grand)) {
        makeRed(grand);
        resolveRed(grand);
    }
}
}

```

Codice 11.15: Metodi per la rimozione nella classe `RBTreemap` (continua dal Codice 11.14).

```

/** Sovrascrive il metodo di TreeMap che viene invocato dopo una rimozione. */
protected void rebalanceDelete(Position<Entry<K,V>> p) {
    if (isRed(p))                                // il nodo rimosso era nero
        makeBlack(p);                            // quindi questo ripristina la profondità nera
    else if (!isRoot(p)) {
        Position<Entry<K,V>> sib = sibling(p);
        if (isInternal(sib) && (isBlack(sib) || isInternal(left(sib))))
            remedyDoubleBlack(p); // il sottoalbero di sib ha altezza nera diversa da 0
    }
}

/** Sistema un'eventuale violazione doppio nero in p (che non è la radice). */
private void remedyDoubleBlack(Position<Entry<K,V>> p) {
    Position<Entry<K,V>> z = parent(p);
    Position<Entry<K,V>> y = sibling(p);
    if (isBlack(y)) {
        if (isRed(left(y)) || isRed(right(y))) { // Caso 1: ristrutturazione a 3 nodi
            Position<Entry<K,V>> x = (isRed(left(y)) ? left(y) : right(y));
            Position<Entry<K,V>> middle = restructure(x);
            setColor(middle, isRed(z)); // usa il vecchio colore di z
            makeBlack(left(middle));
            makeBlack(right(middle));
        } else {                                     // Caso 2: cambio di colorazione
            makeRed(y);
            if (isRed(z))
                makeBlack(z);                      // il problema è risolto
            else if (!isRoot(z))
                remedyDoubleBlack(z);              // propaga il problema
        }
    } else {                                     // Caso 3: ri-orienta un 3-nodo
        rotate(y);
        makeBlack(y);
        makeRed(z);
        remedyDoubleBlack(p);                  // ricomincia la procedura in p
    }
}

```

11.7 Esercizi

Riepilogo e approfondimento

- R-11.1 Inserendo in un albero di ricerca binario inizialmente vuoto le voci $(1, A)$, $(2, B)$, $(3, C)$, $(4, D)$ e $(5, E)$, in questo ordine, che aspetto avrà l'albero?
- R-11.2 Inserire in un albero di ricerca binario vuoto le voci aventi chiavi 30, 40, 24, 58, 48, 26, 11 e 13, in questo ordine. Disegnare l'albero dopo ciascun inserimento.
- R-11.3 Quanti diversi alberi di ricerca binari possono memorizzare le chiavi $\{1, 2, 3\}$?
- R-11.4 Il Dott. Armongus afferma che, dato un insieme di voci da inserire in un albero di ricerca binario, l'ordine in cui avvengono gli inserimenti è ininfluente: si otterrà comunque sempre lo stesso albero. Fornire un piccolo esempio che dimostri che ha torto.
- R-11.5 Il Dott. Armongus afferma che, dato un insieme di voci da inserire in un albero AVL, l'ordine in cui avvengono gli inserimenti è ininfluente: si otterrà comunque sempre lo stesso albero AVL. Fornire un piccolo esempio che dimostri che ha torto.
- R-11.6 La nostra implementazione del metodo ausiliario `treeSearch`, nel Codice 11.3, si basa sulla ricorsione. Per un albero di grandi dimensioni e non bilanciato, è possibile che la pila delle invocazioni usata dall'ambiente Java raggiunga il suo limite per effetto della profondità di ricorsione. Scrivere un'implementazione alternativa di quel metodo che non faccia uso della ricorsione.
- R-11.7 La ristrutturazione di una terna di nodi descritta nella Figura 11.11 si basa su una rotazione singola o doppia? E quella descritta nella Figura 11.13?
- R-11.8 Disegnare l'albero AVL che si ottiene inserendo nell'albero AVL della Figura 11.13b una voce che abbia chiave 52.
- R-11.9 Disegnare l'albero AVL che si ottiene rimuovendo la voce che ha chiave 62 dall'albero AVL della Figura 11.13b.
- R-11.10 Spiegare perché l'esecuzione di una rotazione in un albero binario avente n nodi che usa la rappresentazione basata su array vista nel Paragrafo 8.3.2 richiede un tempo $\Omega(n)$.
- R-11.11 Considerare, in un albero AVL, un'operazione di rimozione che inneschi una ristrutturazione di una terna di nodi relativa al caso in cui i due figli del nodo y hanno la stessa altezza. Seguendo l'esempio della Figura 11.12, disegnare schematicamente l'albero prima e dopo la rimozione. Qual è l'effetto netto dell'operazione sull'altezza del sottoalbero sottoposto a bilanciamento?
- R-11.12 Ripetere l'esercizio precedente, considerando il caso in cui i figli di y hanno inizialmente altezze diverse.
- R-11.13 Le regole per effettuare una rimozione in un albero AVL richiedono espressamente che, quando i due sottoalberi del nodo etichettato come y hanno la stessa altezza, si debba scegliere come x il figlio "allineato" con y (in modo, cioè, che x e y siano entrambi figli sinistri o entrambi figli destri). Per comprendere meglio questo requisito, ripetere l'Esercizio R-11.11 ipotizzando di scegliere il nodo x con il criterio opposto a quello enunciato. Perché, con tale scelta, il ripristino della proprietà dell'albero AVL potrebbe essere problematico?
- R-11.14 Che aspetto ha un albero splay quando si accede alle sue voci in ordine di chiave crescente?

- R-11.15 Eseguire la sequenza di operazioni seguente in un albero splay inizialmente vuoto e disegnarlo dopo ogni insieme di operazioni:
- Inserire le chiavi 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in questo ordine.
 - Cercare le chiavi 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, in questo ordine.
 - Eliminare le chiavi 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in questo ordine.
- R-11.16 Usando un albero splay non si ottengono buone prestazioni per i metodi specifici della mappa ordinata, perché tali metodi non dispongono dell'invocazione del metodo `rebalanceAccess`. Modificare l'implementazione della classe `TreeMap` in modo che tale invocazione sia presente.
- R-11.17 L'albero di ricerca della Figura 11.22a è un albero (2, 4)? Perché, o perché no?
- R-11.18 Per eseguire la suddivisione del nodo w in un albero (2, 4) si può anche, in alternativa alla procedura vista nel capitolo, partizionare w in w' e w'' , in modo che w' sia un 2-nodo e che w'' sia un 3-nodo. Quale chiave, scelta tra k_1, k_2, k_3 e k_4 , va memorizzata nel genitore di w ? Perché?
- R-11.19 Il Dott. Amongus afferma che un albero (2, 4) che memorizzi un determinato insieme di voci avrà sempre la stessa struttura, indipendentemente dall'ordine in cui le voci vengono inserite nell'albero. Dimostrare che ha torto.
- R-11.20 Disegnare quattro diversi alberi rosso-nero che corrispondano allo stesso albero (2, 4).
- R-11.21 Considerare l'insieme di chiavi $K = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$.
- Disegnare un albero (2, 4) che usi gli elementi di K come chiavi, utilizzando il minimo numero di nodi.
 - Disegnare un albero (2, 4) che usi gli elementi di K come chiavi, utilizzando il massimo numero di nodi.
- R-11.22 Data la sequenza di chiavi (5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1), disegnare il risultato che si ottiene inserendo voci aventi tali chiavi (nell'ordine indicato) in:
- Un albero (2, 4) inizialmente vuoto.
 - Un albero rosso-nero inizialmente vuoto.
- R-11.23 Dati i seguenti enunciati relativi agli alberi rosso-nero, fornire una dimostrazione per ciascun enunciato vero e un controesempio per ciascun enunciato falso.
- Un sottoalbero di un albero rosso-nero è un albero rosso-nero.
 - Il fratello di un nodo esterno è un nodo esterno oppure è un nodo rosso.
 - Esiste un unico albero (2, 4) associato a un dato albero rosso-nero.
 - Esiste un unico albero rosso-nero associato a un dato albero (2, 4).
- R-11.24 Dato un albero T che memorizza 100000 voci, qual è il caso peggiore per l'altezza di T nei casi seguenti?
- T è un albero di ricerca binario.
 - T è un albero AVL.
 - T è un albero splay.
 - T è un albero (2, 4).
 - T è un albero rosso-nero.
- R-11.25 Disegnare un esempio di albero rosso-nero che non sia un albero AVL.
- R-11.26 Dimostrare la Proposizione 11.9.
- R-11.27 Dimostrare la Proposizione 11.10.

Creatività

- C-11.28 Spiegare perché effettuando l'attraversamento in ordine simmetrico di un albero di ricerca binario T si visitano le voci contenute sempre nello stesso ordine, indipendentemente dal fatto che T sia gestito come albero AVL, albero splay o albero rosso-nero.
- C-11.29 Spiegare come si possa usare un albero AVL o un albero rosso-nero per ordinare n elementi confrontabili in un tempo $O(n \log n)$ nel caso peggiore.
- C-11.30 Possiamo usare un albero splay per ordinare n elementi confrontabili in un tempo $O(n \log n)$ nel caso peggiore? Spiegare perché, oppure perché no.
- C-11.31 Implementare nella classe `TreeMap` il metodo `putIfAbsent`, così come descritto nell'Esercizio C-10.33.
- C-11.32 Dimostrare che qualsiasi albero binario avente n nodi può essere convertito in qualsiasi altro albero binario avente n nodi usando un numero $O(n)$ di rotazioni.
- C-11.33 Data una chiave k che non è stata trovata all'interno dell'albero di ricerca binario T , dimostrare che tanto la massima chiave minore di k quanto la minima chiave maggiore di k , tra quelle presenti in T , si trovano sul percorso seguito durante la ricerca infruttuosa di k .
- C-11.34 Nel Paragrafo 11.1.4 abbiamo affermato che il metodo `subMap` di un albero di ricerca binario, così come è stato implementato nel Codice 11.6, viene eseguito in un tempo $O(s + h)$, dove s è il numero di voci contenute nella sottomappa restituita dal metodo e h è l'altezza dell'albero. Dimostrare questa affermazione, ragionando sul numero massimo di invocazioni ricorsive del metodo ausiliario `subMapRecurse` che possono avere come parametro posizioni che non appartengono alla sottomappa generata dal metodo.
- C-11.35 Considerare una mappa ordinata implementata mediante un albero di ricerca binario standard, T . Descrivere come si possa eseguire un'operazione `removeSubMap(k_1 , k_2)` che elimini dalla mappa tutte le voci che appartengono alla sottomappa che verrebbe restituita dal metodo `subMap(k_1 , k_2)`, impiegando un tempo $O(s + h)$, dove s è il numero di voci eliminate e h è l'altezza di T .
- C-11.36 Ripetere l'esercizio precedente usando un albero AVL, ottenendo un tempo d'esecuzione $O(s \log n)$. Perché la soluzione del problema precedente, se applicata a un albero AVL, non richiede banalmente un tempo d'esecuzione $O(s + \log n)$?
- C-11.37 Supponiamo di voler definire nel tipo di dato astratto "mappa ordinata" un nuovo metodo, `countRange(k_1 , k_2)`, che determini quante chiavi appartengono all'intervallo delimitato da k_1 e k_2 e che potrebbe essere certamente implementato come adattamento del metodo `subMap`, ottenendo prestazioni $O(s + h)$. Descrivere come si possa modificare la struttura dell'albero di ricerca in modo da ottenere prestazioni $O(h)$ nel caso peggiore per tale metodo `countRange`.
- C-11.38 Se l'approccio descritto nel problema precedente fosse implementato nella classe `TreeMap`, quali ulteriori modifiche sarebbero (eventualmente) necessarie in una sottoclasse come `AVLTreeMap` per gestire le informazioni introdotte a supporto dell'esecuzione del nuovo metodo?
- C-11.39 Disegnare schematicamente un albero AVL tale che una singola operazione `remove` possa richiedere un numero $\Omega(\log n)$ di ristrutturazione di terne di nodi (o di rotazioni), procedendo da una foglia fino alla radice, per ripristinare la proprietà di bilanciamento in altezza.

- C-11.40 Dimostrare che i nodi che diventano temporaneamente sbilanciati in un albero AVL durante un'operazione di inserimento possono essere non consecutivi, ancorché disposti lungo il percorso che va dal nuovo nodo inserito alla radice.
- C-11.41 Dimostrare che, durante l'esecuzione dell'operazione `remove` prevista come standard dal tipo di dato astratto "mappa", nel momento immediatamente successivo all'effettiva rimozione del nodo da rimuovere, in un albero AVL ci può essere al massimo un solo nodo temporaneamente sbilanciato.
- C-11.42 Nella nostra implementazione di AVL, ogni nodo memorizza l'altezza del proprio sottoalbero, che è un numero intero arbitrariamente grande. Lo spazio di memoria usato da un albero AVL può essere ridotto memorizzando, invece, il fattore di bilanciamento di ciascun nodo, che è definito come la differenza tra l'altezza del suo sottoalbero sinistro e l'altezza del suo sottoalbero destro. Di conseguenza, il fatto di bilanciamento di un nodo è sempre uguale a -1 , 0 o $+1$, tranne durante l'esecuzione delle operazioni di inserimento e rimozione, quando può essere temporaneamente uguale anche a -2 o $+2$. Implementare di nuovo la classe `AVLTreeMap` in modo che memorizzi i fattori di bilanciamento anziché le altezze dei sottoalberi.
- C-11.43 Se, in un albero di ricerca binario, memorizziamo un riferimento alla posizione del nodo più a sinistra, l'operazione `firstEntry` può essere eseguita in un tempo $O(1)$. Descrivere come vada modificata l'implementazione degli altri metodi della mappa in modo da tenere aggiornato tale riferimento alla posizione più a sinistra nell'albero.
- C-11.44 Se l'approccio descritto nel problema precedente fosse implementato nella classe `TreeMap`, quali ulteriori modifiche sarebbero (eventualmente) necessarie in una sottoclassificazione come `AVLTreeMap` per gestire l'aggiornamento del riferimento alla posizione più a sinistra?
- C-11.45 Descrivere una possibile modifica della struttura dati "albero di ricerca binario" che fornisca supporto per l'esecuzione in un tempo $O(h)$ delle due seguenti operazioni basate su indice per una mappa ordinata, essendo h l'altezza dell'albero.

atIndex(i): Restituisce la posizione p della voce avente indice i in una mappa ordinata.

indexOf(p): Restituisce l'indice i della voce che si trova nella posizione p in una mappa ordinata.

- C-11.46 Disegnare un albero splay, T_1 , insieme alla sequenza di modifiche che l'ha prodotto, e un albero rosso-nero, T_2 , contenente lo stesso insieme di dieci voci, tali che l'attraversamento in pre-ordine di T_1 visiti le voci nello stesso ordine dell'attraversamento in pre-ordine di T_2 .
- C-11.47 Siano T e U alberi $(2, 4)$ che memorizzano, rispettivamente, n e m voci, tali che tutte le voci di T abbiano chiavi minori delle chiavi di tutte le voci di U . Descrivere un metodo che, in tempo $O(\log n + \log m)$, sia in grado di *unire* T e U in un unico albero che memorizzi tutte le voci di T e di U .
- C-11.48 Sia T un albero rosso-nero che memorizza n voci e sia k la chiave di una voce di T . Spiegare come si possano costruire, a partire da T e in un tempo $O(\log n)$, due alberi rosso-nero T' e T'' , tali che T' contenga tutte le chiavi di T che sono

minori di k e T' contenga tutte le chiavi di T che sono maggiori di k . Questa operazione distrugge T .

- C-11.49 Dimostrare che un albero di ricerca a più vie contenente n voci ha $n + 1$ nodi esterni.
- C-11.50 Quando un albero rosso-nero contiene chiavi che sono tra loro tutte distinte, non è strettamente necessario usare una variabile booleana per contrassegnare ciascun nodo dell'albero come "rosso" o "nero". Descrivere uno schema di implementazione di alberi rosso-nero che non richieda alcuno spazio aggiuntivo nei nodi rispetto a quello usato negli alberi di ricerca binari standard.
- C-11.51 Dimostrare che i nodi di un albero AVL, T , possono essere colorati di "rosso" o di "nero" in modo che T diventi un albero rosso-nero.
- C-11.52 In un albero splay, la procedura standard di estensione richiede due fasi: una fase discendente, per trovare il nodo x da sottoporre a estensione, seguita da una fase ascendente per effettuare effettivamente l'estensione del nodo x . Descrivere un metodo per individuare e estendere x in una sola fase discendente. Ogni sottofase, ora, richiede che si prendano in esame i due nodi successivi nel percorso che scende verso x , con un'eventuale sottofase zig al termine della discesa. Descrivere come vanno eseguite le sottofasi zig-zig, zig-zag e zig.
- C-11.53 Considerare una variante dell'albero splay, detto *albero semi-splay (half-splay tree)*, nel quale la procedura di estensione di un nodo avente profondità d si ferma quando il nodo raggiunge la profondità $\lfloor d/2 \rfloor$. Eseguire un'analisi ammortizzata delle prestazioni degli alberi semi-splay.
- C-11.54 Descrivere una sequenza di accessi alle voci di un albero splay, T , avente n nodi, con n dispari, che trasforma T in un'unica catena di nodi, tale che il percorso che scende all'interno di T si alterni tra figli sinistri e figli destri.

Progettazione

- P-11.55 Implementare di nuovo la classe `TreeMap` usando riferimenti `null` al posto di esplicativi nodi sentinella come foglie dell'albero.
- P-11.56 Modificare l'implementazione della classe `TreeMap` in modo che utilizzi voci *location-aware* (cioè "consapevoli della propria posizione"). Definire i metodi `firstEntry()`, `lastEntry()`, `findEntry(k)`, `before(e)`, `after(e)` e `remove(e)`, in modo che tutti restituiscano un riferimento di tipo `Entry`, tranne l'ultimo, mentre gli ultimi tre devono ricevere un parametro e di tipo `Entry`.
- P-11.57 Effettuare uno studio sperimentale per mettere a confronto la velocità delle nostre implementazioni di albero AVL, albero splay e albero rosso-nero durante l'esecuzione di diverse sequenze di operazioni.
- P-11.58 Ripetere l'esercizio precedente, inserendo la skip list tra le strutture sottoposte a esperimento (si veda l'Esercizio P-10.71).
- P-11.59 Rivedendo il Paragrafo 10.3, implementare il tipo di dato astratto "mappa ordinata" usando un albero (2, 4).
- P-11.60 Scrivere una classe Java che riceva come parametro un albero rosso-nero e lo converta nel suo corrispondente albero (2, 4), oppure riceva un albero (2, 4) e lo converta nel suo corrispondente albero rosso-nero.
- P-11.61 Nel Paragrafo 10.5.3, descrivendo i multi-insiemi e le multi-mappe, abbiamo descritto un approccio generale per adattare una mappa tradizionale in modo

che possa memorizzare tutte le voci aventi chiave duplicata in un contenitore secondario sotto forma di valore di una mappa. Descrivere un'implementazione alternativa di multi-mappa che usi un albero di ricerca binario in modo che ciascuna voce della mappa sia memorizzata in un diverso nodo dell'albero. Con l'esistenza di chiavi duplicate, ridefiniamo la proprietà fondamentale degli alberi di ricerca in modo che tutte le voci del sottoalbero sinistro di una posizione p avente chiave k abbiano chiavi *non maggiori* di k , mentre tutte le voci del sottoalbero destro di p hanno chiavi *non minori* di k . Usare l'interfaccia pubblica definita nel Paragrafo 10.5.3.

- P-11.62 Definire un'implementazione di albero splay che usi la tecnica di estensione che procede soltanto dall'alto verso il basso descritta nell'Esercizio C-11.52. Effettuare un adeguato numero di esperimenti di confronto tra le prestazioni di questo approccio e le prestazioni dell'approccio di estensione standard, che procede dal basso verso l'alto e che è stato implementato in questo capitolo.
- P-11.63 Il tipo di dato astratto *mergeable heap* ("heap che si può fondere") è un'estensione del tipo di dato astratto "coda prioritaria", al quale aggiunge l'operazione *merge(h)* che effettua l'unione dello heap h con lo heap usato per invocare il metodo, inserendo in quest'ultimo tutte le voci di h , che viene svuotato. Descrivere un'implementazione concreta di tale tipo di dato che raggiunga prestazioni $O(\log n)$ per tutte le sue operazioni, essendo n la dimensione dello heap prodotto dall'operazione *merge*.
- P-11.64 "Jumping Leprechauns" è una semplice simulazione di un mondo abitato da n gnomi, numerati da 1 a n , ciascuno dei quali possiede una quantità g_i di oro, per lo gnomo i ; all'inizio della simulazione ogni gnomo possiede oro per un milione di dollari, cioè $g_i = 1000000$ per ogni $i = 1, 2, \dots, n$. Inoltre, la simulazione gestisce anche, per ogni gnomo i , una posizione lungo una retta, rappresentata da un numero in virgola mobile, x_i . Durante ciascun passo della simulazione, tutti gli gnomi vengono esaminati in ordine, per i che va da 1 a n . L'esame di uno gnomo inizia calcolando la sua nuova posizione, usando la formula $x_i = x_i + r g_i$, dove r è un numero casuale in virgola mobile compreso tra -1 e 1. Poi, lo gnomo i ruba metà dell'oro posseduto dallo gnomo più vicino in una delle due direzioni possibili, aggiungendolo al proprio oro, g_i . Scrivere un programma che esegua una serie di passi di questa simulazione per un dato numero di gnomi, n . L'insieme delle posizioni deve essere memorizzato in una mappa ordinata avente una delle strutture descritte in questo capitolo.

Note

Alcune delle strutture dati descritte in questo capitolo sono state trattate in modo approfondito da Knuth nel suo libro *Sorting and Searching* [61], così come da Mehlhorn in [71]. Gli alberi AVL sono stati inventati da Adel'son-Vel'skii e Landis (2) nel 1962. Il libro *Sorting and Searching* di Knuth [61] descrive alberi di ricerca binari, alberi AVL e hashing. L'analisi relativa all'altezza degli alberi di ricerca binari nel caso medio si può trovare nei libri di Aho, Hopcroft e Ullman [6] e di Cormen, Leiserson, Rivest e Stein [25]. Il manuale di Gonnet e Baeza-Yates [38] contiene molti confronti teorici e sperimentali tra diverse implementazioni

di mappa. Aho, Hopcroft e Ullman [5] trattano anche gli alberi (2, 3), che sono simili agli alberi (2, 4). Gli alberi rosso-nero sono stati definiti da Bayer [10]. In un lavoro di Guibas e Sedgewick [42] sono state proposte varianti e proprietà interessanti degli alberi rosso-nero. Il lettore interessato a conoscere alberi bilanciati di tipo diverso viene rimandato ai libri di Mehlhorn [71] e Tarjan [88], oltre a un capitolo del libro di Mehlhorn e Tsakalidis [73]. Il libro di Knuth più volte citato [61] è un'ulteriore lettura d'eccezione, che presenta i primi esempi di alberi bilanciati. Gli alberi splay sono stati inventati da Sleator e Tarjan [83] (e si veda anche [88]).

12

Ordinamento e selezione

12.1 Ordinamento per fusione (*merge-sort*)

Abbiamo già presentato diversi algoritmi di ricerca, tra i quali l'ordinamento per inserimento (*insertion-sort*, nei Paragrafi 3.1.2, 7.6 e 9.4.1), l'ordinamento per selezione (*selection-sort*, nel Paragrafo 9.4.1), l'ordinamento a bolle (*bubble-sort*, nell'Esercizio C-7.51) e l'ordinamento mediante heap (*heap-sort*, nel Paragrafo 9.4.2). In questo capitolo presenteremo altri quattro algoritmi di ordinamento (*merge-sort*, *quick-sort*, *bucket-sort* e *radix-sort*), poi discuteremo i loro vantaggi e svantaggi con un confronto, nel Paragrafo 12.4.

12.1.1 Dividi-e-conquista (*divide-and-conquer*)

I primi due algoritmi che descriviamo in questo capitolo, *merge-sort* e *quick-sort*, usano la ricorsione in uno schema per la progettazione di algoritmi chiamato **dividi-e-conquista** (*divide-and-conquer*). Nel Capitolo 5 abbiamo già visto come la ricorsione sia potente per descrivere algoritmi in modo elegante, e lo schema dividi-e-conquista è costituito dalle tre fasi qui elencate:

1. *Dividi*. Se la dimensione dei dati da elaborare è minore di una determinata soglia (diciamo, uno o due elementi), risolvi il problema direttamente usando un metodo banale e restituisci la soluzione così ottenuta, altrimenti suddividi i dati in due o più sottoinsiemi disgiunti.

2. **Conquista.** Risovi ricorsivamente i sotto-problemi associati a sottoinsiemi di dati.
3. **Combina.** Prendi le soluzioni dei sotto-problemi e fondile insieme per generare la soluzione del problema originario.

Ordinamento mediante la strategia *dividi-e-conquista*

Per prima cosa descriviamo l'algoritmo di ordinamento per fusione (*merge-sort*) ad alto livello, senza preoccuparci del fatto che i dati da ordinare siano memorizzati in un array o in una lista concatenata (vedremo presto implementazioni concrete di entrambi i casi). Per ordinare una sequenza S di n elementi usando le tre fasi della strategia dividi-e-conquista, l'algoritmo di ordinamento per fusione procede in questo modo:

1. **Dividi.** Se S è vuota o ha un solo elemento, restituisci immediatamente S , perché è una sequenza già ordinata. Altrimenti (S ha almeno due elementi), elimina tutti gli elementi da S e memorizzali in due sequenze, S_1 e S_2 , ciascuna delle quali conterrà "circa" metà degli elementi di S : nello specifico, S_1 contiene i primi $\lfloor n/2 \rfloor$ elementi di S e S_2 contiene i rimanenti $\lceil n/2 \rceil$ elementi.
2. **Conquista.** Ordina ricorsivamente S_1 e S_2 .
3. **Combina.** Memorizza nuovamente gli elementi in S fondendo le sequenze ordinate S_1 e S_2 in una sequenza ordinata.

Con riferimento alla fase di divisione, ricordiamo che la notazione $\lfloor x \rfloor$ indica il più grande numero intero non maggiore di x (la cosiddetta funzione *floor*) e, analogamente, la notazione $\lceil x \rceil$ indica il più piccolo numero intero non minore di x (la cosiddetta funzione *ceiling*).

Possiamo visualizzare l'esecuzione dell'algoritmo merge-sort tramite un albero binario T , chiamato *albero merge-sort*. Ogni nodo v di T rappresenta una delle invocazioni ricorsive dell'algoritmo merge-sort e a v viene associata la sequenza S elaborata, appunto, dall'invocazione rappresentata da v . I figli del nodo v sono associati alle invocazioni ricorsive che elaborano le sotto-sequenze S_1 e S_2 di S . I nodi esterni di T sono, invece, associati ai singoli elementi di S , corrispondenti alle esecuzioni dell'algoritmo che non fanno uso di ulteriori invocazioni ricorsive.

La Figura 12.1 riassume un'esecuzione dell'algoritmo di ordinamento per fusione mostrando le sequenze iniziali e finali elaborate da ciascun nodo dell'albero merge-sort. Le Figure 12.2, 12.3 e 12.4 mostrano, invece, l'evoluzione di questo stesso albero passo dopo passo.

La visualizzazione dell'algoritmo mediante l'albero merge-sort ci aiuta nell'analisi del tempo d'esecuzione dell'algoritmo stesso. In particolare, dato che la dimensione delle sequenze viene circa dimezzata in conseguenza di ciascuna invocazione ricorsiva, l'altezza dell'albero merge-sort è circa $\log n$ (ricordando che, quando omettiamo la base dei logaritmi, si intende utilizzata la base 2).

Proposizione 12.1: L'albero merge-sort associato a un'esecuzione dell'algoritmo di ordinamento per fusione applicato a una sequenza di dimensione n ha altezza $\lceil \log n \rceil$.

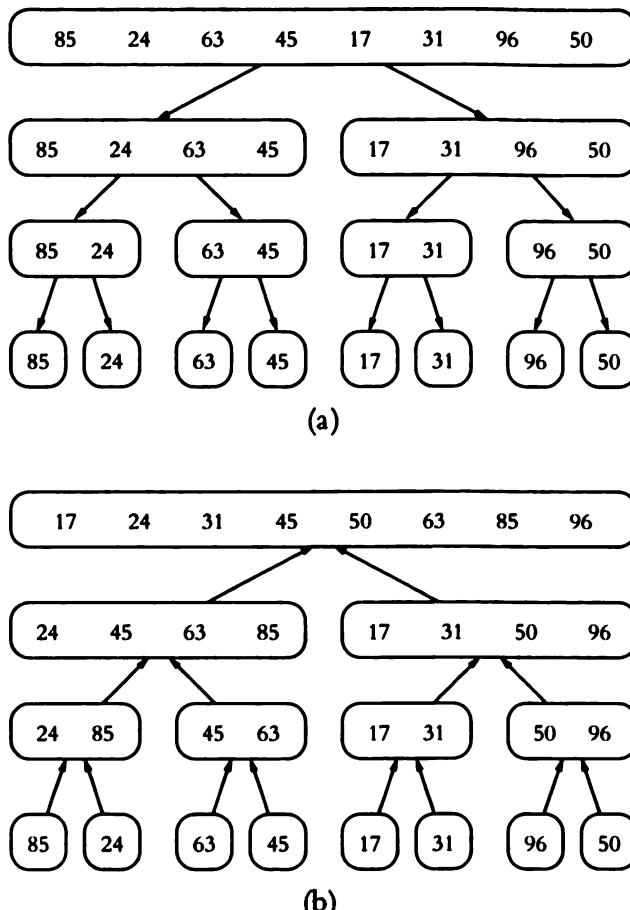


Figura 12.1: L'albero merge-sort T corrispondente all'esecuzione dell'algoritmo di ordinamento per fusione su una sequenza di 8 elementi: (a) sequenze elaborate da ciascun nodo di T , prima della loro elaborazione; (b) sequenze elaborate da ciascun nodo di T , dopo l'elaborazione.

Lasciamo la dimostrazione della Proposizione 12.1 come semplice esercizio (R-12.1) e la useremo per analizzare il tempo d'esecuzione dell'algoritmo di ordinamento per fusione.

Dopo aver visto una panoramica, anche grafica, del funzionamento dell'ordinamento per fusione, consideriamo più in dettaglio i singoli passi di questo algoritmo dividere-e-conquistare. La divisione di una sequenza di dimensione n richiede una sua separazione in corrispondenza dell'elemento avente indice $\lceil n/2 \rceil$, dopodiché può iniziare l'esecuzione delle invocazioni ricorsive che ricevono come parametri le due sequenze più corte. La fase difficile è l'ultima, che deve combinare le due sequenze ordinate in un'unica sequenza ordinata. Quindi, prima di presentare l'analisi dell'ordinamento per fusione, dobbiamo dire qualcosa di più in merito a come possa essere eseguita questa fase.

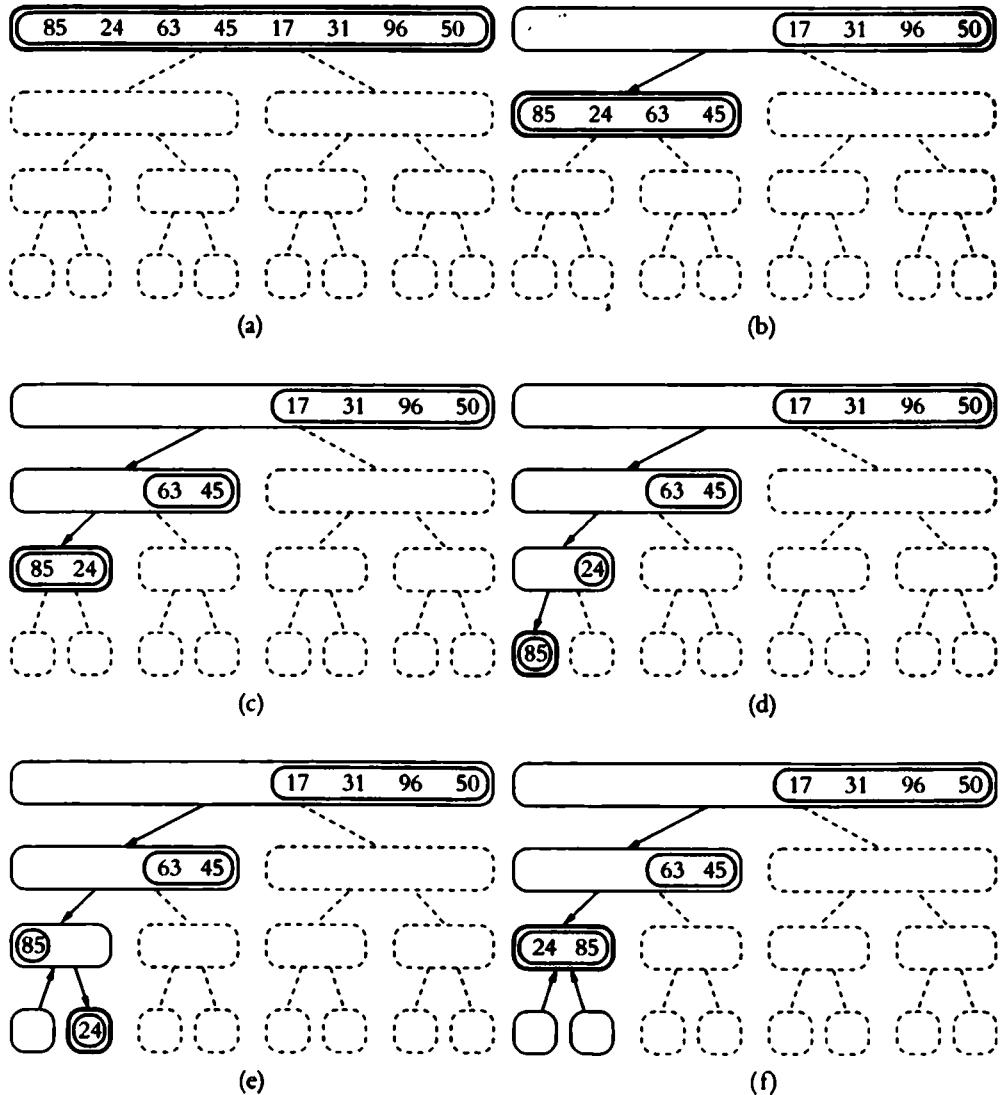


Figura 12.2: Visualizzazione (che prosegue nelle Figure 12.3 e 12.4) di un'esecuzione dell'algoritmo di ordinamento per fusione. Ogni nodo dell'albero rappresenta un'invocazione ricorsiva dell'algoritmo. I nodi disegnati con linea tratteggiata rappresentano invocazioni che non sono ancora iniziate. Il nodo evidenziato con tratto più spesso è quello in esecuzione. I nodi vuoti, disegnati con tratto più sottile, rappresentano invocazioni terminate. Gli altri nodi (disegnati con tratto sottile ma non vuoti) rappresentano invocazioni che sono in attesa che un figlio restituisca il valore elaborato.

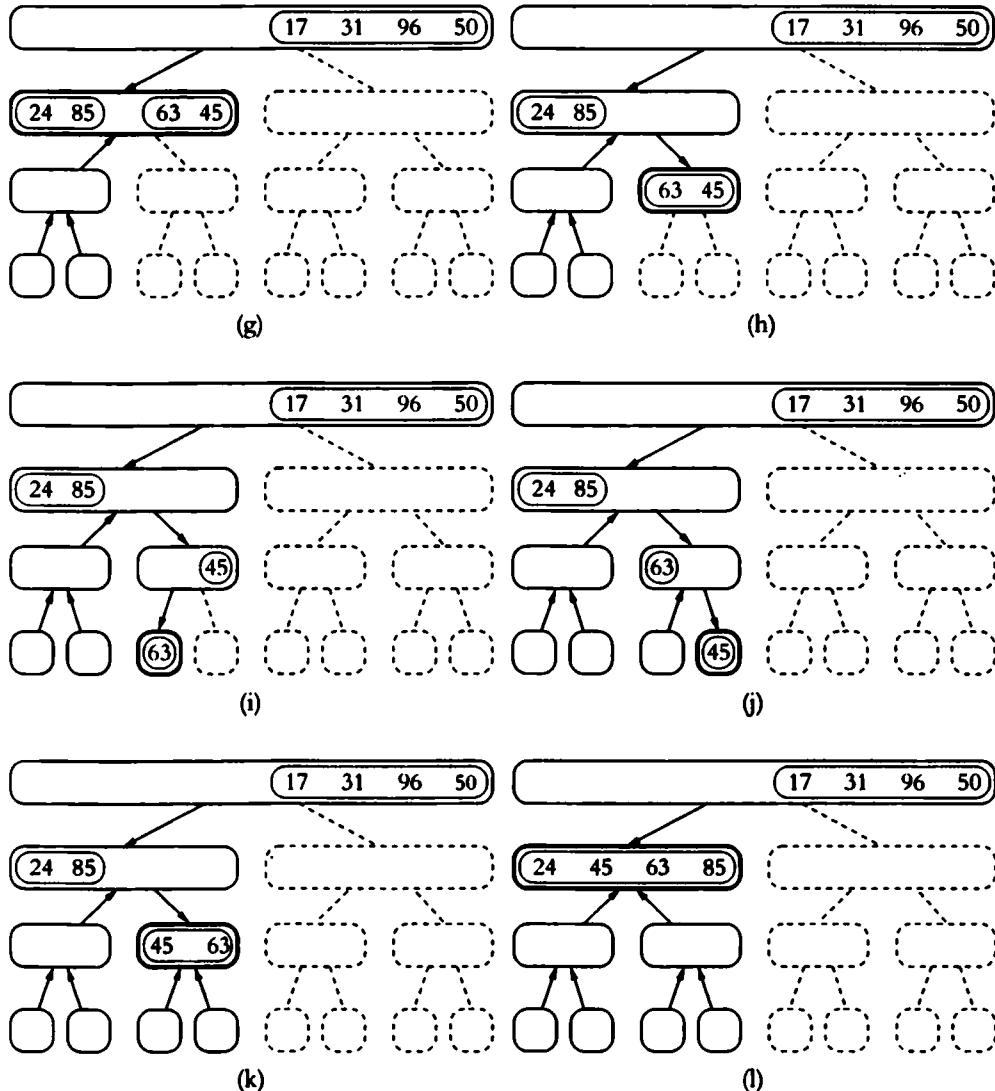


Figura 12.3: Visualizzazione di un'esecuzione dell'algoritmo di ordinamento per fusione (continua dalla Figura 12.3 e prosegue nella Figura 12.4).

12.1.2 Implementazione di merge-sort basata su array

Iniziamo concentrandoci sul caso in cui una sequenza di dati sia rappresentata mediante un array. Il metodo merge (delineato nel Codice 12.1) ha il compito di risolvere il sotto-problema della fusione di due sequenze ordinate in precedenza, S_1 e S_2 , memorizzando in S la sequenza ordinata prodotta. Copiamo un elemento durante ogni iterazione del ciclo **while**, determinando ogni volta se l'elemento vada prelevato da S_1 o da S_2 . L'algoritmo completo di ordinamento per fusione, secondo la strategia dividi-e-conquistà, è presentato nel Codice 12.2.

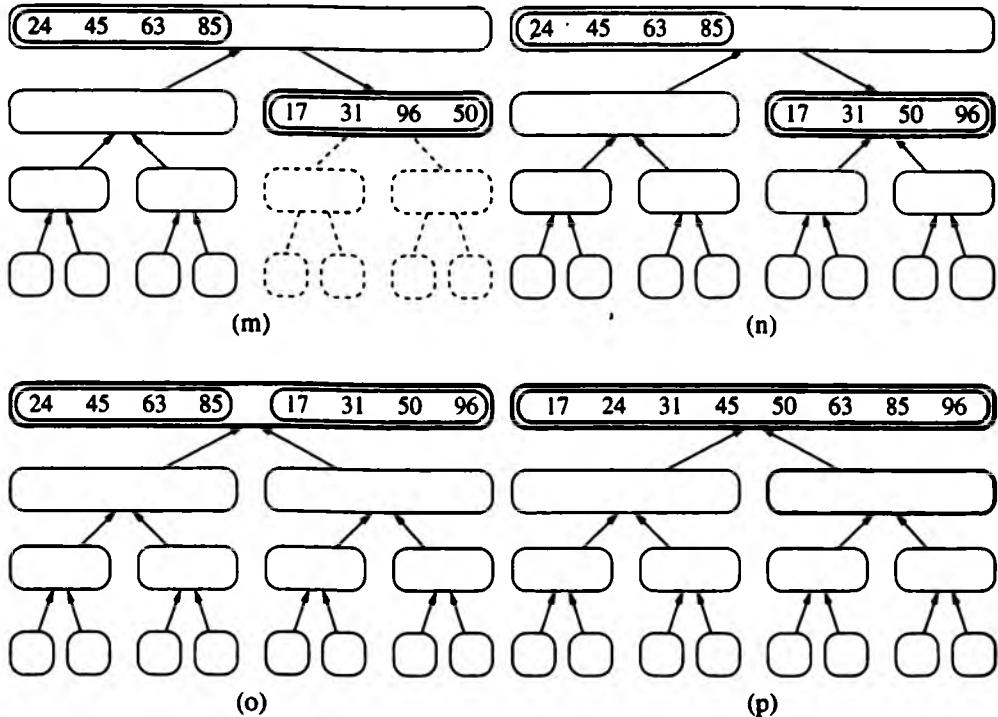


Figura 12.4: Visualizzazione di un'esecuzione dell'algoritmo di ordinamento per fusione (continua dalle Figure 12.3 e 12.4). Tra il passo (m) e il passo (n) sono state omesse alcune invocazioni. Si noti la fusione finale che avviene al passo (p).

Nella Figura 12.5 abbiamo illustrato un passo della procedura di fusione. Durante tale procedura, l'indice i rappresenta il numero di elementi di S_1 che sono stati copiati in S , mentre l'indice j rappresenta il numero di elementi di S_2 che sono stati copiati in S . Nell'ipotesi che sia S_1 sia S_2 abbiano ancora almeno un elemento che non è stato copiato, copiamo il più piccolo dei due elementi in esame. Dato che in precedenza sono stati copiati $i + j$ elementi, il prossimo viene memorizzato in $S[i + j]$ (ad esempio, quando $i + j$ vale 0, l'elemento viene copiato in $S[0]$). Quando arriviamo alla fine di una delle due sequenze, dobbiamo copiare ordinatamente gli elementi rimasti nell'altra.

Codice 12.1: Un'implementazione in Java dell'operazione di fusione per un array.

```

1  /** Fonde il contenuto degli array S1 e S2 nell'array S di dimensione opportuna. */
2  public static <K> void merge(K[] S1, K[] S2, K[] S, Comparator<K> comp) {
3      int i = 0, j = 0;
4      while (i + j < S.length) {
5          if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
6              S[i+j] = S1[i++]; // copia l'i-esimo elemento di S1 e incrementa i
7          else
8              S[i+j] = S2[j++]; // copia il j-esimo elemento di S2 e incrementa j
9      }
10 }
```

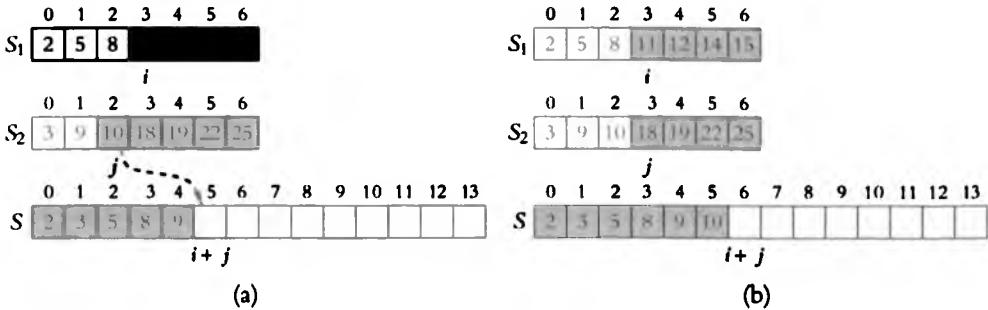


Figura 12.5: Un passo della fusione di due array ordinati, nel caso in cui $S_2[j] < S_1[i]$. La figura (a) mostra gli array prima della copiatura, mentre nella figura (b) la copiatura è già avvenuta.

Codice 12.2: Un'implementazione in Java dell'algoritmo ricorsivo di ordinamento per fusione applicato a un array, usando il metodo `merge` definito nel Codice 12.1.

```

1  /** Ordinamento per fusione del contenuto dell'array S. */
2  public static <K> void mergeSort(K[] S, Comparator<K> comp) {
3      int n = S.length;
4      if (n < 2) return;      // l'array è così banale che è già ordinato
5      // dividi
6      int mid = n/2;
7      K[] S1 = Arrays.copyOfRange(S, 0, mid); // copia la prima metà
8      K[] S2 = Arrays.copyOfRange(S, mid, n); // copia la seconda metà
9      // conquista (mediante ricorsione)
10     mergeSort(S1, comp);        // ordina la copia della prima metà di S
11     mergeSort(S2, comp);        // ordina la copia della seconda metà di S
12     // fondi i risultati
13     merge(S1, S2, S, comp);    // fondi le metà ordinate, memorizzandole in S
14 }
```

Osserviamo che i metodi `merge` e `mergeSort` si basano sull'utilizzo di un esemplare di `Comparator` per confrontare una coppia di oggetti generici, che si presume appartengano a un insieme totalmente ordinato: è lo stesso approccio che abbiamo visto nella definizione delle code prioritarie, nel Paragrafo 9.2.2, e nello studio dell'implementazione delle mappe ordinate, nei Capitoli 10 e 11.

12.1.3 Il tempo d'esecuzione di merge-sort

Cominciamo analizzando il tempo d'esecuzione dell'algoritmo `merge`. Siano, rispettivamente, n_1 e n_2 il numero di elementi di S_1 e S_2 . È evidente che le operazioni eseguite all'interno di ciascuna iterazione del ciclo `while` richiede un tempo $O(1)$. Poi, vediamo l'osservazione chiave: durante ciascuna iterazione del ciclo, un elemento viene copiato in S , prelevandolo da S_1 o da S_2 , dopodiché quell'elemento non viene più preso in esame. Di conseguenza, il numero di iterazioni del ciclo è $n_1 + n_2$ e, quindi, il tempo d'esecuzione dell'algoritmo `merge` è $O(n_1 + n_2)$.

Dopo aver così analizzato il tempo d'esecuzione dell'algoritmo `merge`, utilizzato per combinare le soluzioni dei sotto-problemi, analizziamo il tempo d'esecuzione dell'intero algoritmo di ordinamento per fusione, nell'ipotesi di dover ordinare una sequenza iniziale

di n elementi. Per semplicità, poi, restringiamo il campo dell'analisi, ipotizzando che n sia una potenza di 2, lasciando per esercizio (R-12.3) la dimostrazione del fatto che il risultato della nostra analisi vale anche quando n non è una potenza di 2.

Nel valutare la ricorsione dell'algoritmo merge-sort, ci basiamo sulla tecnica di analisi vista nel Paragrafo 5.2: sommiamo il tempo speso in ciascuna invocazione ricorsiva, escludendo il tempo trascorso in attesa che le invocazioni iniziate terminino. Nel caso del metodo `mergeSort`, teniamo conto del tempo speso per dividere la sequenza in due sotto-sequenze e per eseguire l'invocazione di `merge`, usata alla fine per combinare le due sequenze ordinate, ma escludiamo il tempo richiesto per portare a termine le due invocazioni ricorsive di `mergeSort`.

Un albero merge-sort, come quello rappresentato nelle Figure 12.2, 12.3 e 12.4, può guidare la nostra analisi. Consideriamo l'invocazione ricorsiva associata a un nodo v dell'albero merge-sort T . La fase di divisione nel nodo v è semplice: viene eseguita in un tempo proporzionale alla dimensione della sequenza associata a v , creando copie delle due metà della sequenza. Inoltre, abbiamo già osservato che anche la fase di fusione richiede un tempo proporzionale alla dimensione della sequenza generata dalla fusione stessa. Se indichiamo con i la profondità del nodo v , il tempo speso nel nodo v è $O(n/2^i)$, perché la dimensione della sequenza elaborata dall'invocazione ricorsiva associata al nodo v è proprio uguale a $n/2^i$.

Osservando l'albero T più globalmente, come si può vedere nella Figura 12.6, vediamo che, data la nostra definizione di "tempo speso in un nodo", il tempo d'esecuzione dell'algoritmo di ordinamento per fusione è uguale alla somma dei tempi spesi nei nodi di T . Osserviamo che T ha esattamente 2^i nodi di profondità i : questa semplice osservazione ha una conseguenza importante, perché implica che il tempo totale speso in tutti i nodi di T aventi profondità i è $O(2^i \cdot n/2^i)$, cioè $O(n)$. Per la Proposizione 12.1, l'altezza di T è $\lceil \log n \rceil$, quindi, dal momento che il tempo speso in ciascuno dei livelli di T è $O(n)$ e il numero di tali livelli è $\lceil \log n \rceil + 1$, abbiamo il risultato seguente.

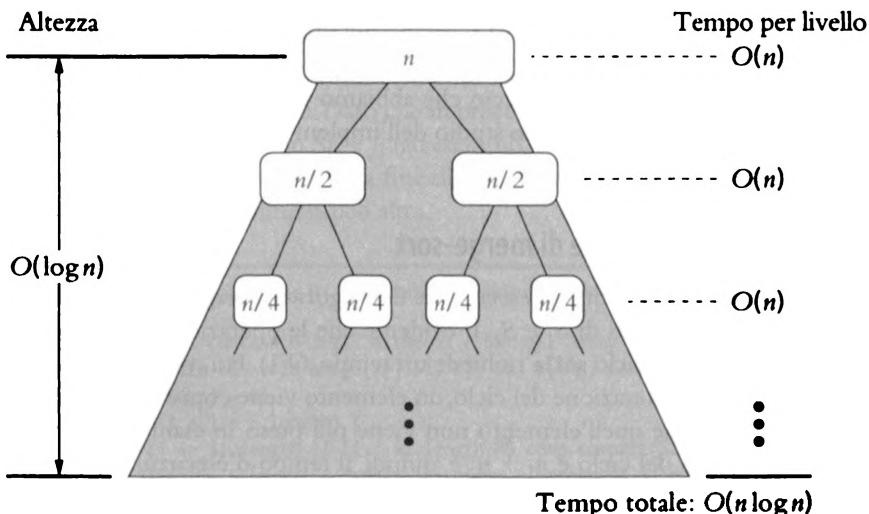


Figura 12.6: Un'analisi grafica del tempo d'esecuzione dell'ordinamento per fusione. Ogni nodo dell'albero rappresenta il tempo speso in una specifica invocazione ricorsiva ed è etichettato con la dimensione del suo sotto-problema.

Proposizione 12.2: L'algoritmo di ordinamento per fusione ordina una sequenza S di dimensione n in un tempo $O(n \log n)$, nell'ipotesi che due elementi di S possano essere confrontati tra loro in un tempo $O(1)$.

12.1.4 Merge-sort ed equazioni di ricorrenza *

C'è un altro modo per dimostrare che il tempo d'esecuzione dell'algoritmo di ordinamento per fusione è $O(n \log n)$, cioè per dimostrare la Proposizione 12.2. Nello specifico, possiamo lavorare in modo più diretto con la natura ricorsiva dell'algoritmo: in questo paragrafo presenteremo tale analisi del tempo d'esecuzione di merge-sort e, nel farlo, introdurremo il concetto matematico di **equazione di ricorrenza** (o di **relazione di ricorrenza**).

Indichiamo con la funzione $t(n)$ il tempo d'esecuzione nel caso peggiore dell'algoritmo merge-sort applicato a una sequenza di dimensione n . Dato che merge-sort è ricorsivo, possiamo caratterizzare la funzione $t(n)$ mediante un'equazione in cui la funzione $t(n)$ viene espressa ricorsivamente in termini di se stessa. Per semplificare la nostra caratterizzazione di $t(n)$, limitiamoci al caso in cui n è una potenza di 2 (lasciando come esercizio la dimostrazione che la caratterizzazione asintotica ottenuta valga anche nel caso generale). Con questa ipotesi, possiamo definire $t(n)$ come:

$$t(n) = \begin{cases} b & \text{se } n \leq 1 \\ 2t(n/2) + cn & \text{altrimenti} \end{cases}$$

Un'espressione come questa è chiamata **equazione di ricorrenza**, perché la funzione definita compare sia a sinistra sia a destra del segno di uguaglianza. Sebbene tale caratterizzazione sia corretta e accurata, quello che vogliamo avere è una caratterizzazione di $t(n)$ in forma O-grande, che non richieda l'utilizzo della funzione $t(n)$ stessa, cioè vogliamo una caratterizzazione di $t(n)$ in **forma chiusa**.

Possiamo ottenere una soluzione in forma chiusa di questa equazione di ricorrenza applicando ripetutamente l'equazione stessa, nell'ipotesi che n sia sufficientemente grande. Ad esempio, dopo un'ulteriore applicazione dell'equazione, possiamo scrivere una nuova ricorrenza che definisce $t(n)$ in questo modo:

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2t(n/2^2) + 2(cn/2) + cn = 2^2t(n/2^2) + 2cn. \end{aligned}$$

Se applichiamo l'equazione di nuovo, otteniamo $t(n) = 2^3t(n/2^3) + 3cn$. A questo punto dovremmo riuscire a vedere uno schema che si pone in evidenza, perché, dopo aver applicato questa equazione i volte, otteniamo:

$$t(n) = 2^it(n/2^i) + icn.$$

Il problema che rimane, ora, è quello di capire quando porre fine a questa procedura. Per farlo ricordiamo che, quando $n \leq 1$, si passa alla forma chiusa $t(n) = b$, cosa che accadrà quando $2^i = n$, cioè quando $i = \log n$. Facendo questa sostituzione, otteniamo:

$$\begin{aligned}
 t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n)cn \\
 &= nt(1) + cn\log n \\
 &= nb + cn\log n.
 \end{aligned}$$

Abbiamo, quindi, ottenuto una dimostrazione alternativa del fatto che $t(n)$ sia $O(n \log n)$.

12.1.5 Implementazioni alternative di merge-sort

Ordinare liste concatenate

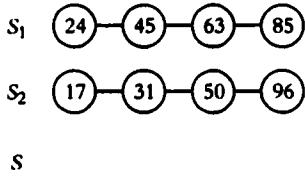
L'algoritmo di ordinamento per fusione può essere facilmente adattato per utilizzare qualsiasi forma di contenitore che somigli a una coda. Nel Codice 12.3 presentiamo una di tali implementazioni, basata sull'utilizzo della classe `LinkedQueue` vista nel Paragrafo 6.2.3. Il limite $O(n \log n)$ visto nella Proposizione 12.2 per merge-sort si applica anche a questa implementazione, perché ciascuna operazione elementare viene comunque eseguita in un tempo $O(1)$ anche usando una lista concatenata. Nella Figura 12.7 si può vedere un esempio di esecuzione di questa versione dell'algoritmo `merge`.

Codice 12.3: Un'implementazione di merge-sort usando una normale coda.

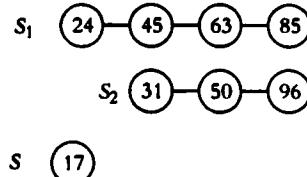
```

1  /** Fonde il contenuto delle code ordinate S1 e S2 nella coda vuota S. */
2  public static <K> void merge(Queue<K> S1, Queue<K> S2, Queue<K> S,
3                                Comparator<K> comp) {
4      while (!S1.isEmpty() && !S2.isEmpty()) {
5          if (comp.compare(S1.first(), S2.first()) < 0)
6              S.enqueue(S1.dequeue()); // prendi il prossimo elemento da S1
7          else
8              S.enqueue(S2.dequeue()); // prendi il prossimo elemento da S2
9      }
10     while (!S1.isEmpty())
11         S.enqueue(S1.dequeue()); // sposta tutti gli elementi rimasti in S1
12     while (!S2.isEmpty())
13         S.enqueue(S2.dequeue()); // sposta tutti gli elementi rimasti in S2
14 }
15
16 /** Ordinamento per fusione del contenuto della coda S. */
17 public static <K> void mergeSort(Queue<K> S, Comparator<K> comp) {
18     int n = S.size();
19     if (n < 2) return; // la coda è così banale che è già ordinata
20     // dividi
21     Queue<K> S1 = new LinkedQueue<K>(); // o qualsiasi altra implementazione di coda
22     Queue<K> S2 = new LinkedQueue<K>();
23     while (S1.size() < n/2)
24         S1.enqueue(S.dequeue()); // sposta in S1 i primi n/2 elementi di S
25     while (!S.isEmpty())
26         S2.enqueue(S.dequeue()); // sposta in S1 gli elementi rimasti in S
27     // conquista (mediante ricorsione)
28     mergeSort(S1, comp); // ordina la copia della prima metà di S
29     mergeSort(S2, comp); // ordina la copia della seconda metà di S
30     // fondi i risultati
31     merge(S1, S2, S, comp); // fonda le metà ordinate, memorizzandole in S
32 }

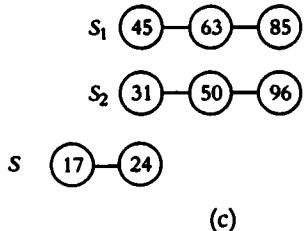
```



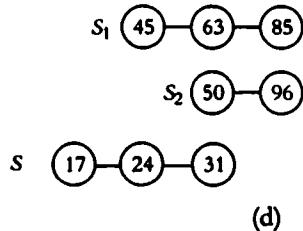
(a)



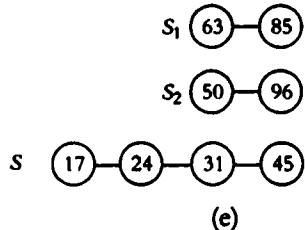
(b)



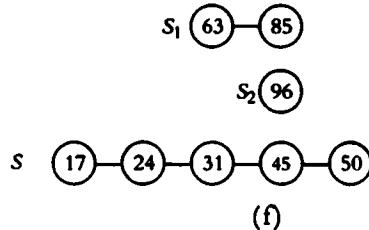
(c)



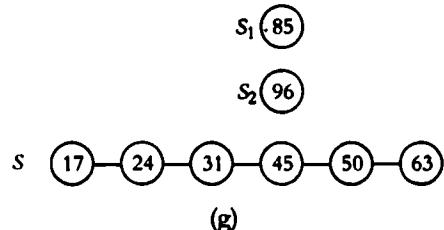
(d)



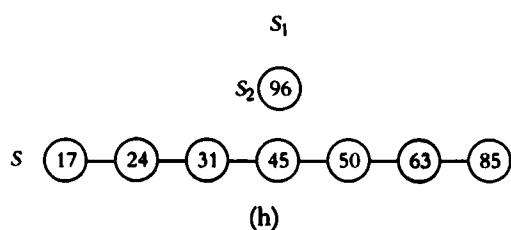
(e)



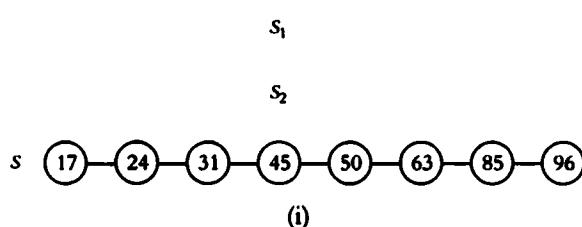
(f)



(g)



(h)



(i)

Figura 12.7: Esempio di esecuzione dell'algoritmo merge, così come implementato nel Codice 12.3, cioè usando code per memorizzare gli elementi da ordinare e ordinati, oltre che per la memorizzazione temporanea.

Un'implementazione *bottom-up* (e non ricorsiva) di merge-sort

Esiste anche una versione non ricorsiva dell'ordinamento per fusione basato su array, che viene eseguita in un tempo $O(n \log n)$. In pratica è un po' più veloce del merge-sort ricorsivo, perché evita il tempo che viene speso per la gestione delle invocazioni ricorsive e risparmia la memoria temporanea relativa a ciascun livello dell'albero merge-sort. L'idea principale è quella di eseguire l'ordinamento procedendo, nell'albero merge-sort, dal basso verso l'alto, cioè in modalità *bottom-up*, con fusioni eseguite livello per livello risalendo, appunto, l'albero merge-sort. Dato un array di elementi da ordinare, iniziamo fondendo ogni coppia di elementi consecutivi, generando sequenze ordinate di lunghezza due, una per ogni coppia. Poi, fondiamo queste sequenze in sequenze di lunghezza quattro, fondendo queste ultime in sequenze di lunghezza otto, e così via, fino a quando l'intero array non è stato ordinato. Per contenere l'occupazione di spazio in memoria a un valore ragionevole, usiamo un secondo array che memorizza tutte le sequenze ottenute temporaneamente dalle fusioni intermedie (scambiando il ruolo di array di input e array di output dopo ogni iterazione del ciclo principale). Nel Codice 12.4 presentiamo un'implementazione in Java di questa strategia, usando il metodo di libreria `System.arraycopy` per copiare un insieme di celle consecutive da un array all'altro. Un approccio simile, sempre *bottom-up*, può essere utilizzato anche per ordinare liste concatenate, come si vedrà nell'Esercizio C-12.30.

Codice 12.4: Un'implementazione dell'algoritmo non ricorsivo di ordinamento per fusione.

```
1  /** Fonde in[start..start+inc-1] e in[start+inc..start+2*inc-1] scrivendo out[ ]. */
2  public static <K> void merge(K[] in, K[] out, Comparator<K> comp,
3                                int start, int inc) {
4      int end1 = Math.min(start + inc, in.length);    // fine della porzione 1
5      int end2 = Math.min(start + 2 * inc, in.length); // fine della porzione 2
6      int x=start;                                     // indice nella porzione 1
7      int y=start+inc;                                // indice nella porzione 2
8      int z=start;                                     // indice nell'array out
9      while (x < end1 && y < end2)
10         if (comp.compare(in[x], in[y]) < 0)
11             out[z++] = in[x++];                      // prendi il prossimo dalla porzione 1
12         else
13             out[z++] = in[y++];                      // prendi il prossimo dalla porzione 2
14     if (x < end1) System.arraycopy(in, x, out, z, end1-x); // copia il resto di 1
15     else if (x < end2) System.arraycopy(in, y, out, z, end2-x); // copia il resto di 2
16 }
17 /** Ordinamento per fusione del contenuto dell'array dato. */
18 public static <K> void mergeSortBottomUp(K[] orig, Comparator<K> comp) {
19     int n = orig.length;
20     K[] src = orig;                                // altro nome dell'array originario
21     K[] dest = (K[]) new Object[n]; // crea un nuovo array temporaneo
22     K[] temp;                                     // riferimento usato solo per gli scambi
23     for (int i=1; i < n; i *= 2) { // ogni iterazione ordina tutti gli array lunghi i
24         for (int j=0; j < n; j += 2*i) // ogni passo fonde due array di lunghezza i
25             merge(src, dest, comp, j, i);
26         temp = src; src = dest; dest = temp; // scambia i ruoli degli array
27     }
28     if (orig != src)
29         System.arraycopy(src, 0, orig, 0, n); // se serve, copia il risultato in orig
30 }
```

12.2 Ordinamento quick-sort

Il prossimo algoritmo di ordinamento che presentiamo è chiamato *quick-sort* (cioè “ordinamento veloce”). Come merge-sort, anche questo algoritmo è basato sul paradigma *dividi-e-conquista*, ma usa tale tecnica in un modo che potremmo definire opposto, perché la parte complicata dell’elaborazione viene svolta *prima* delle invocazioni ricorsive.

Descrizione ad alto livello di quick-sort

L’algoritmo quick-sort ordina una sequenza S usando un semplice approccio ricorsivo. L’idea di base è quella di applicare la tecnica dividi-e-conquista: si divide S in sotto-sequenze, si esegue una ricorsione per ordinare ciascuna sotto-sequenza e, poi, si combinano le sotto-sequenze ordinate mediante semplice concatenazione. Più in dettaglio, con riferimento alla Figura 12.8, l’algoritmo quick-sort è costituito dalle tre fasi seguenti:

1. **Dividi.** Se S è vuota o ha un solo elemento, restituisci immediatamente S , perché è una sequenza già ordinata. Altrimenti (S ha almeno due elementi), seleziona un elemento x in S , che sarà chiamato *pivot*. Spesso la scelta del pivot cade sull’ultimo elemento di S . Elimina da S tutti i suoi elementi e memorizzali in una di queste tre sequenze:
 - L , che memorizza gli elementi di S che sono minori (*less than*, da cui la lettera L) di x
 - E , che memorizza gli elementi di S che sono uguali (*equal*) a x
 - G , che memorizza gli elementi di S che sono maggiori (*greater than*) di x
 Ovviamente, se gli elementi di S sono tutti distinti, allora E conterrà un solo elemento: il pivot stesso.
2. **Conquista.** Ordina ricorsivamente le sequenze L e G .
3. **Combina.** Memorizza nuovamente gli elementi in S ordinatamente, inserendo prima gli elementi di L , poi quelli di E e, infine, quelli di G .

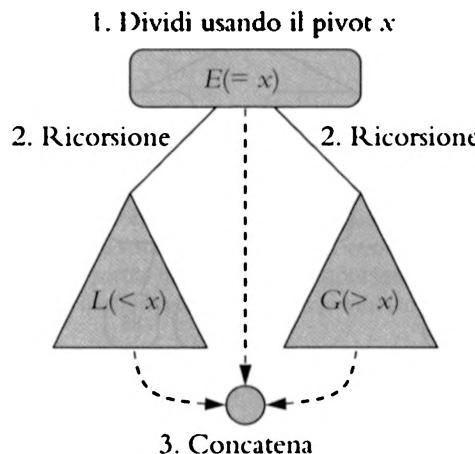
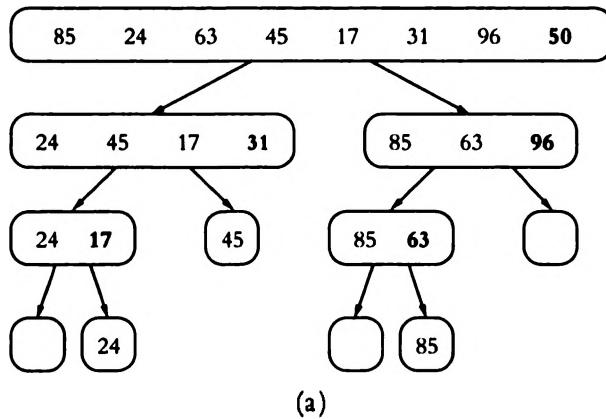


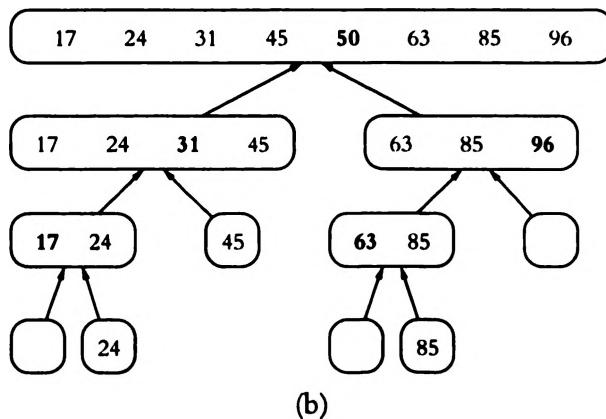
Figura 12.8: Una rappresentazione grafica schematica dell’algoritmo quick-sort.

Come abbiamo visto con merge-sort, anche l'esecuzione di quick-sort può essere visualizzata mediante un albero di ricorsione binario, chiamato *albero quick-sort*. La Figura 12.9 riassume un'esecuzione dell'algoritmo quick-sort mostrando le sequenze elaborate da ciascun nodo dell'albero quick-sort, rappresentate prima dell'elaborazione nella Figura 12.9a e dopo l'elaborazione nella Figura 12.9b. L'evoluzione, passo dopo passo, dell'albero quick-sort è visibile nelle Figure 12.10, 12.11 e 12.12.

Diversamente da merge-sort, però, l'altezza dell'albero quick-sort associato a un'esecuzione dell'algoritmo è, nel caso peggiore, lineare. Questo accade, ad esempio, se la sequenza da ordinare è costituita da n elementi distinti già ordinati: in questo caso, la scelta standard, che prevede di prendere sempre come pivot l'ultimo elemento della sotto-sequenza da ordinare, porta la prima volta a una sotto-sequenza L di dimensione $n - 1$, mentre la sotto-sequenza E ha dimensione 1 e la sotto-sequenza G ha dimensione 0. Per effetto di ogni successiva invocazione ricorsiva di quick-sort sulla sotto-sequenza L , la dimensione diminuisce di un'unità, quindi l'altezza dell'albero quick-sort è $n - 1$.



(a)



(b)

Figura 12.9: Albero quick-sort T per un'esecuzione dell'algoritmo quick-sort applicato a una sequenza di 8 elementi: (a) sequenze da elaborare in corrispondenza di ciascun nodo di T ; (b) sequenze elaborate da ciascun nodo di T . Il pivot scelto in ciascun nodo è evidenziato in grassetto.

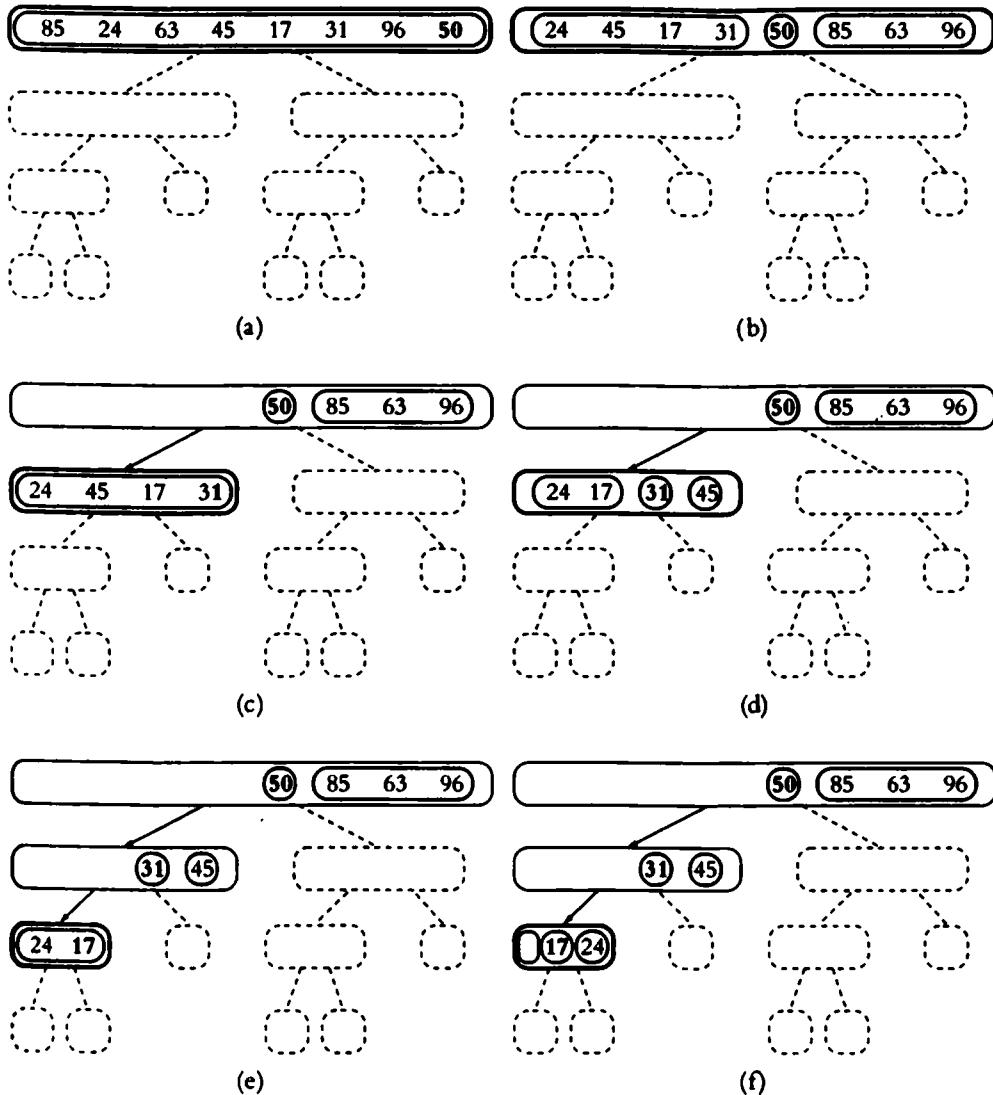


Figura 12.10: Visualizzazione (che prosegue nelle Figure 12.11 e 12.12) di un'esecuzione dell'algoritmo quick-sort. Ogni nodo dell'albero rappresenta un'invocazione ricorsiva dell'algoritmo. I nodi disegnati con linea tratteggiata rappresentano invocazioni che non sono ancora iniziate. Il nodo evidenziato con tratto più spesso è quello in esecuzione. I nodi vuoti, disegnati con tratto più sottile, rappresentano invocazioni terminate. Gli altri nodi (disegnati con tratto sottile ma non vuoti) rappresentano invocazioni che sono in attesa che un figlio restituisca il valore elaborato. Si osservino, in particolare, le fasi di divisione eseguite in (b), (d) e (f).

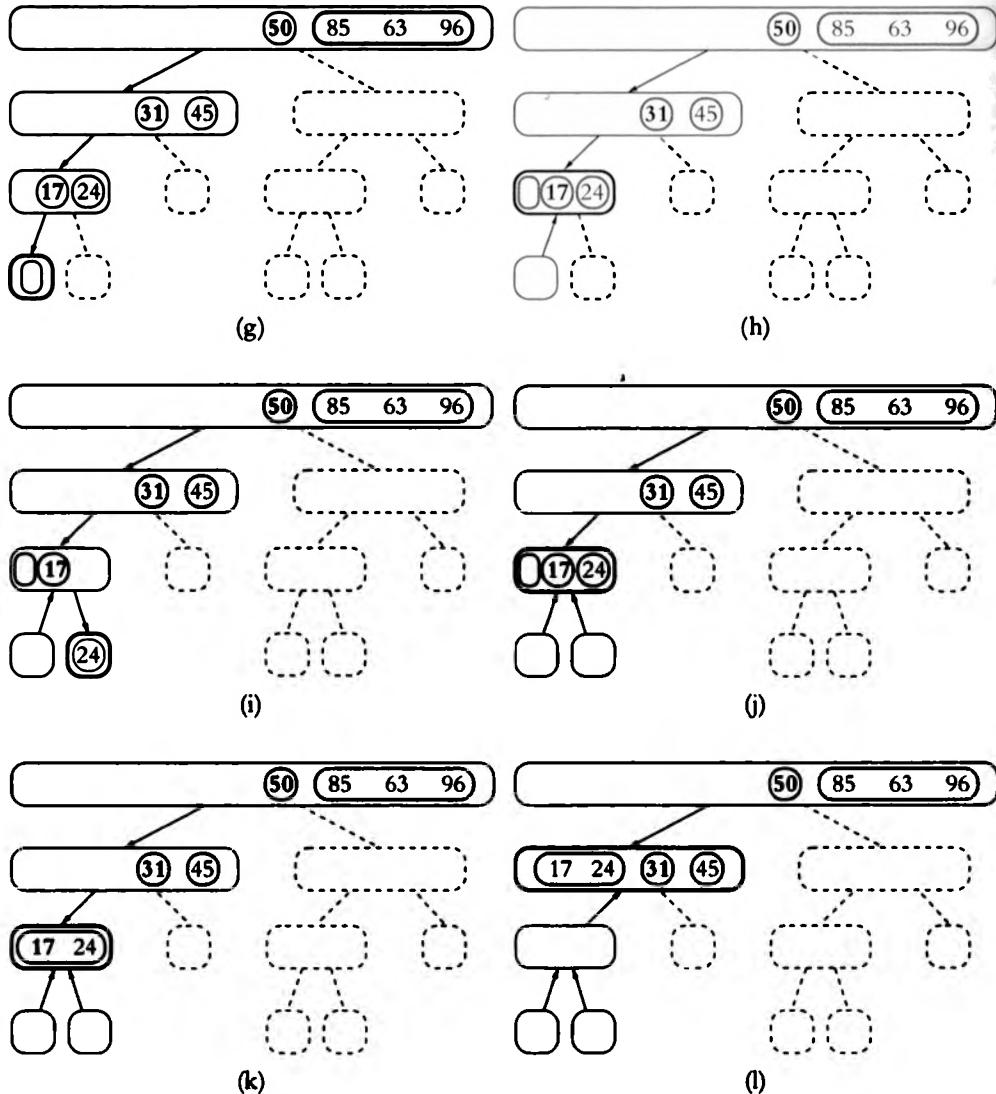


Figura 12.11: Visualizzazione di un'esecuzione dell'algoritmo quick-sort (continua dalla Figura 12.10 e prosegue nella Figura 12.12). Si osservi, in particolare, la concatenazione eseguita in (k).

Eseguire quick-sort per sequenze generiche

Nel Codice 12.5 presentiamo un'implementazione dell'algoritmo quick-sort che opera con sequenze di qualunque tipo che funzionino come una coda. Questa specifica versione usa esemplari della classe `LinkedQueue` vista nel Paragrafo 6.2.3, mentre nel Paragrafo 12.2.2 vedremo un'implementazione più sintetica di quick-sort che usa sequenze memorizzate in array.

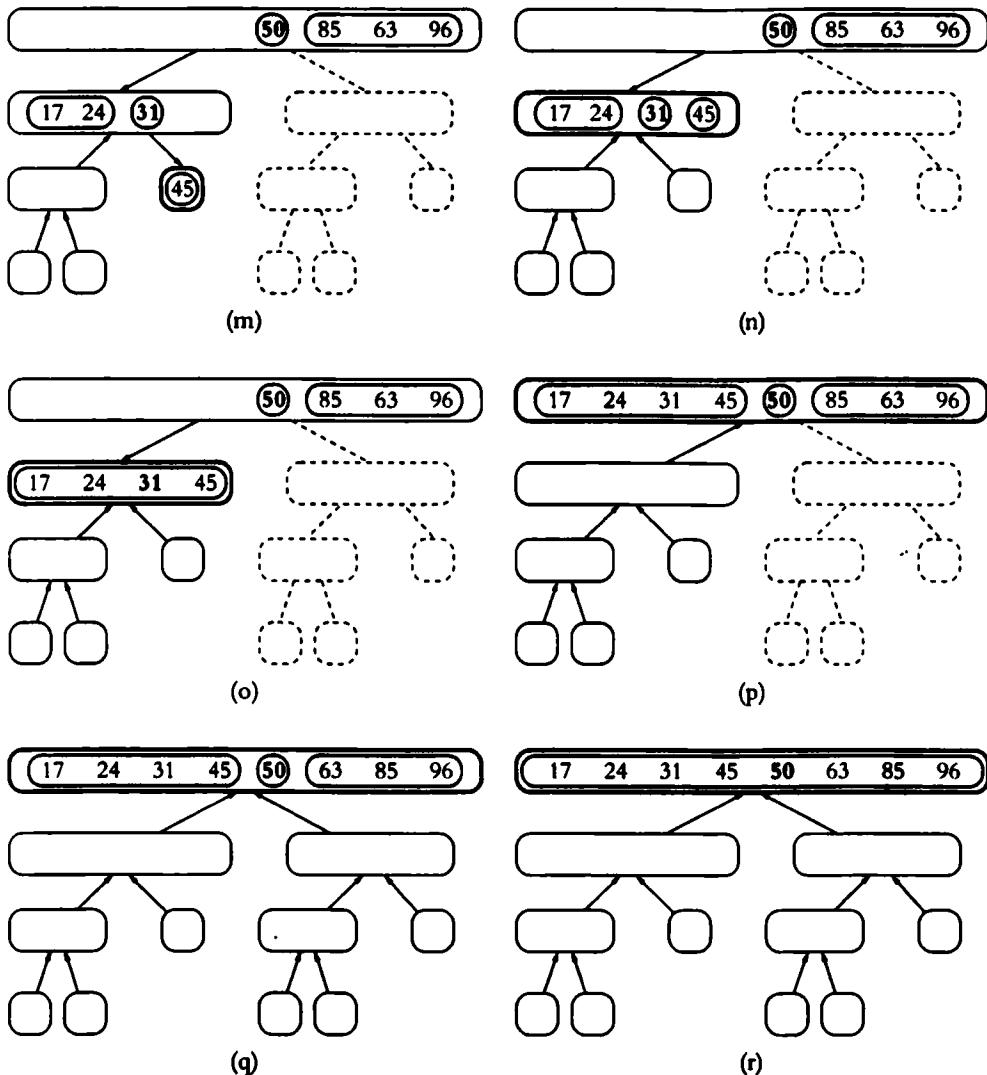


Figura 12.12: Visualizzazione di un'esecuzione dell'algoritmo quick-sort (continua dalle Figure 12.10 e 12.11). Tra il passo (p) e il passo (q) sono state omesse alcune invocazioni. Si notino le fasi di concatenazione in (o) e in (r).

Questa nostra implementazione sceglie come pivot il primo elemento della coda (perché è quello più facilmente accessibile), poi divide la sequenza nelle tre code L , E e G contenenti, rispettivamente, gli elementi minori del pivot, quelli uguali al pivot e quelli maggiori del pivot. Quindi, viene eseguita la ricorsione per ordinare le code L e G , trasferendo infine gli elementi dalle code ordinate L , E e G alla coda originaria S , che era rimasta vuota dopo la fase di divisione. Tutte le operazioni sulle code vengono eseguite in un tempo $O(1)$ nel caso peggiore, se queste sono implementate con una lista concatenata, come nella classe `LinkedQueue`.

Codice 12.5: Quick-sort per una sequenza S implementata con una coda.

```

1  /** Quick-sort che ordina il contenuto della coda S. */
2  public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
3      int n = S.size();
4      if (n < 2) return;          // la coda è così banale che è già ordinata
5      // dividi
6      K pivot = S.first();     // usa arbitrariamente il primo elemento come pivot
7      Queue<K> L = new LinkedQueue<K>();
8      Queue<K> E = new LinkedQueue<K>();
9      Queue<K> G = new LinkedQueue<K>();
10     while (!S.isEmpty()) { // divide la coda originaria in L, E e G
11         K element = S.dequeue();
12         int c = comp.compare(element, pivot);
13         if (c < 0)             // element è minore del pivot
14             L.enqueue(element);
15         else if (c == 0)       // element è uguale al pivot
16             E.enqueue(element);
17         else                  // element è maggiore del pivot
18             G.enqueue(element);
19     }
20     // conquista (mediante ricorsione)
21     quickSort(L, comp);      // ordina gli elementi minori del pivot
22     quickSort(G, comp);      // ordina gli elementi maggiori del pivot
23     // concatena i risultati
24     while (!L.isEmpty())
25         S.enqueue(L.dequeue());
26     while (!E.isEmpty())
27         S.enqueue(E.dequeue());
28     while (!G.isEmpty())
29         S.enqueue(G.dequeue());
30 }

```

Tempo d'esecuzione di quick-sort

Possiamo analizzare il tempo d'esecuzione di quick-sort usando la stessa tecnica utilizzata nel Paragrafo 12.1.3 per merge-sort. Nello specifico, dobbiamo caratterizzare il tempo speso in ciascun nodo dell'albero quick-sort T e sommare tale tempo per tutti i nodi.

Esaminando il Codice 12.5, vediamo che la fase iniziale di divisione e la fase conclusiva di concatenazione di quick-sort possono essere realizzate in un tempo lineare, quindi il tempo speso in un nodo v di T è proporzionale alla *dimensione* $s(v)$ dei dati da elaborare nel nodo v , definita come la dimensione della sequenza ricevuta come parametro dall'invocazione di quick-sort associata al nodo v . Dato che la sotto-sequenza E contiene almeno un elemento (il pivot), la somma delle dimensioni associate ai figli di v è al massimo uguale a $s(v) - 1$.

Indichiamo con s_i la somma delle dimensioni delle sequenze da elaborare associate ai nodi aventi profondità i in un albero quick-sort T . Ovviamente, $s_0 = n$, perché la radice r di T è associata all'intera sequenza da ordinare. Poi, $s_i \leq n - 1$, perché il pivot non viene propagato ai figli di r . Più in generale, deve sempre essere $s_i < s_{i-1}$, perché gli elementi delle sotto-sequenze assegnate a nodi di profondità i provengono tutte da sotto-sequenze distinte assegnate a nodi di profondità $i - 1$, e almeno un elemento di *ciascuna* sequenza di profondità $i - 1$ non viene propagato alla profondità i perché appartiene a una sequenza E .

Possiamo, quindi, limitare superiormente il tempo di un'esecuzione di quick-sort a una funzione $O(n \cdot h)$, dove h è l'altezza complessiva dell'albero quick-sort T relativo a tale

esecuzione dell'algoritmo. Sfortunatamente, nel caso peggiore l'altezza di un albero quick-sort è $n - 1$, come già osservato nel Paragrafo 12.2, quindi quick-sort ha, nel caso peggiore, un tempo d'esecuzione $O(n^2)$. Paradossalmente, se scegliessimo come pivot sempre l'ultimo elemento della sequenza in esame, questo comportamento di caso peggiore si avrebbe per quegli esemplari del problema in cui l'ordinamento dovrebbe essere più semplice: quando la sequenza da ordinare è, in realtà, già ordinata.

Dato il nome di questo algoritmo (ricordiamo che *quick* significa *veloce*), ci saremmo aspettati di scoprire che quick-sort viene eseguito velocemente, cosa che effettivamente, in pratica, avviene spesso. Il caso migliore di quick-sort applicato a una sequenza di elementi distinti si verifica quando le sotto-sequenze L e G hanno approssimativamente la stessa dimensione: in tal caso, come già visto per merge-sort, l'albero ha altezza $O(\log n)$ e, quindi, il tempo d'esecuzione di quick-sort è $O(n \log n)$, affermazione che verrà dimostrata nell'Esercizio R-12.12. Di più, possiamo osservare che si ottiene un tempo $O(n \log n)$ anche se la suddivisione tra L e G non è perfetta: ad esempio, se ogni fase di suddivisione generasse una sotto-sequenza che contiene un quarto degli elementi (e l'altra ne contiene tre quarti), l'altezza dell'albero rimarrebbe $O(\log n)$ e, quindi, le prestazioni complessive sarebbero comunque $O(n \log n)$.

Nel prossimo paragrafo vedremo come l'introduzione della casualità nella scelta del pivot renda il comportamento di quick-sort essenzialmente uguale, in media, a quello appena descritto, con un tempo d'esecuzione atteso $O(n \log n)$.

12.2.1 Quick-sort con scelta casuale del pivot

Una possibile strategia per analizzare quick-sort prevede di ipotizzare che il pivot divida sempre la sequenza in esame in un modo ragionevolmente bilanciato, ma una tale ipotesi presuppone una conoscenza sulla distribuzione dei valori da ordinare che tipicamente non è disponibile. Ad esempio, dovremmo ipotizzare che ci si trovi raramente di fronte a sequenze da ordinare che siano "quasi" ordinate, cosa che, invece, è piuttosto frequente in molte applicazioni. Fortunatamente, questa ipotesi non è necessaria per dimostrare la nostra intuizione relativa al comportamento di quick-sort.

In generale, vorremmo riuscire in qualche modo ad avvicinarci al tempo d'esecuzione di quick-sort che ne caratterizza il caso migliore. Per avvicinarsi al caso migliore, ovviamente, il pivot deve dividere la sequenza da elaborare, S , in porzioni di dimensioni quasi uguali: se questo accadesse, il tempo d'esecuzione sarebbe asintoticamente uguale a quello del caso migliore. In pratica, la scelta di pivot che siano ogni volta vicini al "centro" dell'insieme degli elementi porta a un tempo d'esecuzione $O(n \log n)$ per quick-sort.

Scelta casuale del pivot

Dato che l'obiettivo della fase di partizionamento dell'algoritmo quick-sort è quello di dividere la sequenza S in modo sufficientemente bilanciato, introduciamo casualità nell'algoritmo e scegliamo come pivot un *elemento a caso* della sequenza: invece di scegliere come pivot il primo o l'ultimo elemento di S , scegliamo un elemento di S a caso, senza modificare nessun altro aspetto dell'algoritmo. Questa variante di quick-sort è chiamata *randomized quick-sort* (cioè "quick-sort reso casuale" o *probabilistico*) e l'affermazione seguente dimostra che il tempo d'esecuzione atteso dell'algoritmo quick-sort probabilistico applicato a una sequenza di n elementi è $O(n \log n)$. Questo calcolo del tempo atteso deriva dalla media

eseguita su tutte le possibili scelte casuali fatte dall'algoritmo ed è indipendente da qualunque ipotesi sulla distribuzione dei dati nella sequenza da ordinare che viene fornita all'algoritmo.

Proposizione 12.3: *Il tempo atteso per l'esecuzione dell'algoritmo quick-sort probabilistico su una sequenza S di dimensione n è $O(n \log n)$.*

Dimostrazione: Sia S una sequenza di n elementi e sia T l'albero binario associato all'esecuzione dell'algoritmo quick-sort probabilistico applicato a S . Innanzitutto, osserviamo che il tempo d'esecuzione dell'algoritmo è proporzionale al numero di confronti eseguiti. Consideriamo l'invocazione ricorsiva associata a un nodo di T e osserviamo che, durante l'esecuzione dell'invocazione, tutti i confronti avvengono tra l'elemento pivot e un altro elemento presente nella (sotto-)sequenza che deve essere ordinata dall'invocazione stessa. Quindi, possiamo calcolare il numero di confronti eseguiti dall'algoritmo usando la formula $\sum_{x \in S} C(x)$, dove $C(x)$ è il numero di confronti che riguardano l'elemento x diverso dal pivot. Nel seguito dimostreremo che, per ogni elemento $x \in S$, il valore atteso di $C(x)$ è $O(\log n)$. Dato che il valore atteso di una somma è uguale alla somma dei valori attesi dei suoi addendi, un limite superiore $O(\log n)$ per il valore atteso di $C(x)$ per qualsiasi x implica che il tempo d'esecuzione atteso per quick-sort probabilistico sia $O(n \log n)$.

Per dimostrare che il valore atteso di $C(x)$ è $O(\log n)$ per qualsiasi x prendiamo un elemento x arbitrario e consideriamo il percorso di nodi dell'albero T associati alle invocazioni ricorsive per le quali x appartiene alla sequenza elaborata, come si può vedere nella Figura 12.13. Per definizione, $C(x)$ è uguale alla lunghezza di tale percorso, perché x prenderà parte, come elemento diverso dal pivot, a un confronto per ogni livello dell'albero finché non viene scelto come pivot oppure diventa l'unico elemento rimasto nella sequenza.

Indichiamo con n_d la dimensione della sequenza elaborata dal nodo del percorso avente profondità d nell'albero T , con $0 \leq d \leq C(x)$. Dato che tutti gli elementi fanno parte dell'invocazione iniziale, $n_0 = n$. Sappiamo che la dimensione della sequenza, per ciascuna invocazione ricorsiva, è minore di almeno un'unità rispetto alla dimensione dell'invocazione associata al genitore del nodo che la rappresenta, quindi $n_{d+1} \leq n_d - 1$ per ogni $d < C(x)$. Nel caso peggiore, questo implica che $C(x) \leq n - 1$, perché la procedura ricorsiva termina quando $n_d = 1$ oppure x viene scelto come pivot.

A questo punto possiamo dimostrare un'affermazione più forte: se a ogni livello dell'albero il pivot viene scelto in modo casuale, il valore atteso di $C(x)$ è $O(\log n)$. La scelta del pivot di profondità d lungo il percorso citato è considerata "buona" se $n_{d+1} \leq 3n_d/4$. La scelta di un pivot sarà buona con probabilità almeno $1/2$, perché ci sono almeno $n_d/2$ elementi nella sequenza in esame che, se scelti come pivot, faranno in modo che almeno $n_d/4$ elementi vengano assegnati a ciascuno dei due sotto-problemi, lasciando così x in un gruppo avente al massimo $3n_d/4$ elementi.

Concludiamo osservando che, prima che x rimanga da solo, il numero di tali scelte "buone" che verranno effettuate è, al massimo, uguale a $\log_{4/3} n$. Dato che una scelta è buona con probabilità almeno $1/2$, il numero atteso di invocazioni ricorsive che vengono effettuate prima di poter fare un numero di scelte buone del pivot uguale a $\log_{4/3} n$ è al massimo $2 \log_{4/3} n$, da cui discende che $C(x)$ è $O(\log n)$. ■

Con un'analisi più rigorosa, si può dimostrare che il tempo d'esecuzione dell'algoritmo quick-sort probabilistico è $O(n \log n)$ con elevata probabilità (si veda l'Esercizio C-12.55).

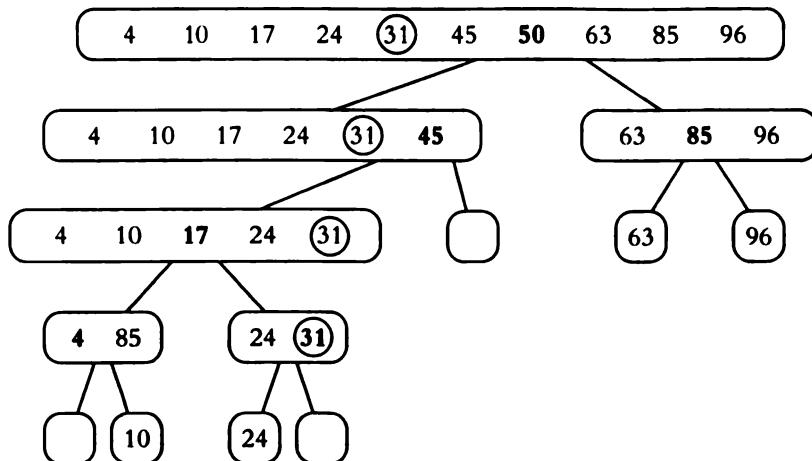


Figura 12.13: Un esempio che illustra l'analisi discussa per la Proposizione 12.3, relativa a un'esecuzione di quick-sort probabilistico. Ci concentriamo sull'elemento $x = 31$, per il quale $C(x) = 3$, perché, come elemento diverso dal pivot, viene confrontato con tre elementi: 50, 45 e 17. In base alla notazione introdotta, si ha $n_0 = 10$, $n_1 = 6$, $n_2 = 5$ e $n_3 = 2$, e i numeri 50 e 17, come pivot, sono scelte "buone".

12.2.2 Ulteriori ottimizzazioni per quick-sort

Un algoritmo opera *sul posto* (*in-place*) se usa soltanto una piccola quantità di memoria aggiuntiva rispetto a quella necessaria per memorizzare i dati da elaborare e i dati da presentare in uscita. La nostra implementazione di heap-sort, nel Paragrafo 9.4.2, è un esempio di algoritmo di ordinamento che opera sul posto, mentre l'implementazione di quick-sort che abbiamo presentato nel Codice 12.5 non lo è, perché usa i contenitori aggiuntivi L , E e G durante la suddivisione di S , all'interno di ogni invocazione ricorsiva. L'algoritmo quick-sort, quando viene applicato a una sequenza memorizzata in un array, può essere adattato a funzionare "sul posto" e proprio tale ottimizzazione viene solitamente utilizzata nella maggior parte delle implementazioni.

L'esecuzione dell'algoritmo quick-sort sul posto richiede, però, un po' di astuzia, perché bisogna usare la sequenza ricevuta inizialmente per memorizzare le sotto-sequenze usate in tutte le invocazioni ricorsive. Nel Codice 12.6 vediamo, quindi, l'algoritmo `quickSortInPlace`, che esegue l'algoritmo quick-sort sul posto per ordinare un array. L'algoritmo modifica la sequenza ricevuta usando scambi tra elementi e non crea sotto-sequenze in modo esplicito: una sotto-sequenza è rappresentata implicitamente da un intervallo di posizioni, specificato dall'indice più a sinistra, a , e dall'indice più a destra, b . La fase di divisione viene eseguita scandendo l'array simultaneamente nei due sensi, usando le variabili locali `left`, che procede da sinistra a destra, e `right`, che procede a ritroso, scambiando tra loro le coppie di elementi che sono in ordine inverso rispetto a quello previsto: il tutto è mostrato nella Figura 12.14. Non c'è alcuna esplicita fase di "concatenazione", perché la concatenazione delle due sotto-sequenze è implicita nell'uso "sul posto" dell'array originario.

Codice 12.6: Quick-sort "sul posto" per un array S. L'intero array viene ordinato invocando quickSortInPlace(S, comp, 0, S.length-1).

```
1  /** Ordina la porzione di array S[a..b], estremi inclusi. */
2  public static <K> void quickSortInPlace(K[] S, Comparator<K> comp
3                                              int a, int b) {
4      if (a >= b) return;          // la porzione di array è così banale che è già ordinata
5      int left = a;
6      int right = b-1;
7      K pivot = S[b];
8      K temp;                    // variabile temporanea, usata per gli scambi
9      while (left <= right) {
10          // va avanti finché raggiunge right oppure un valore non minore del pivot
11          while (left <= right && comp.compare(S[left], pivot) < 0) left++;
12          // va indietro finché raggiunge left oppure un valore non maggiore del pivot
13          while (left <= right && comp.compare(S[right], pivot) > 0) right--;
14          if (left <= right) { // gli indici non si sono raggiunti
15              // quindi scambia i valori e restringi l'intervallo
16              temp = S[left]; S[left] = S[right]; S[right] = temp;
17              left++; right--;
18          }
19      }
20      // metti il pivot nella sua posizione definitiva (individuata da left)
21      temp = S[left]; S[left] = S[b]; S[b] = temp;
22      // esegui le invocazioni ricorsive
23      quickSortInPlace(S, comp, a, left - 1);
24      quickSortInPlace(S, comp, left + 1, b);
25  }
```

Se una sequenza contiene valori duplicati, è bene notare che non creiamo esplicitamente tre sotto-sequenze, L , E e G , come invece facevamo nella descrizione originaria di quick-sort, quindi gli elementi uguali al pivot (al di là del pivot stesso) vanno a finire, dove capita, all'interno delle due sotto-sequenze. L'Esercizio R-12.11 indaga sui dettagli relativi al comportamento della nostra implementazione in presenza di elementi duplicati, mentre l'Esercizio C-12.34 descrive una variante dell'algoritmo, operante sul posto, che suddivide effettivamente la sequenza in esame in tre sotto-sequenze, L , E e G .

Sebbene l'implementazione che abbiamo appena descritto suddivida gli elementi in due sotto-sequenze operando sul posto, osserviamo che l'algoritmo quick-sort completo ha bisogno di spazio per gestire una pila di esecuzione (*runtime stack*) di dimensioni proporzionali alla profondità dell'albero di ricorsione, che, in questo caso, può avere altezza massima $n - 1$, anche se, a dire il vero, la profondità attesa di tale pila è $O(\log n)$, che è un valore basso, se confrontato con n . C'è, poi, un piccolo trucco che ci può garantire che la dimensione di questa pila sia proprio $O(\log n)$: l'idea consiste nel progettare una versione non ricorsiva dell'algoritmo quick-sort sul posto, usando una pila esplicita per elaborare iterativamente i sotto-problemi (ciascuno dei quali può essere rappresentato da una coppia di indici che segnala l'inizio e la fine di un sotto-array). Ogni iterazione prevede di estrarre dalla pila il sotto-problema che si trova in cima, sottoponendolo alla fase di divisione (se ha dimensione sufficiente) e impilando i due nuovi sotto-problemi così generati. Il trucco è questo: quando si impilano i nuovi sotto-problemi, si impila per primo quello di dimensioni maggiori. In questo modo, le dimensioni dei sotto-problemi diventano almeno il doppio ogni volta che scendiamo lungo la pila; quindi, la pila può avere una profondità al massimo $O(\log n)$. I dettagli di questa implementazione sono lasciati come esercizio (P-12.59).

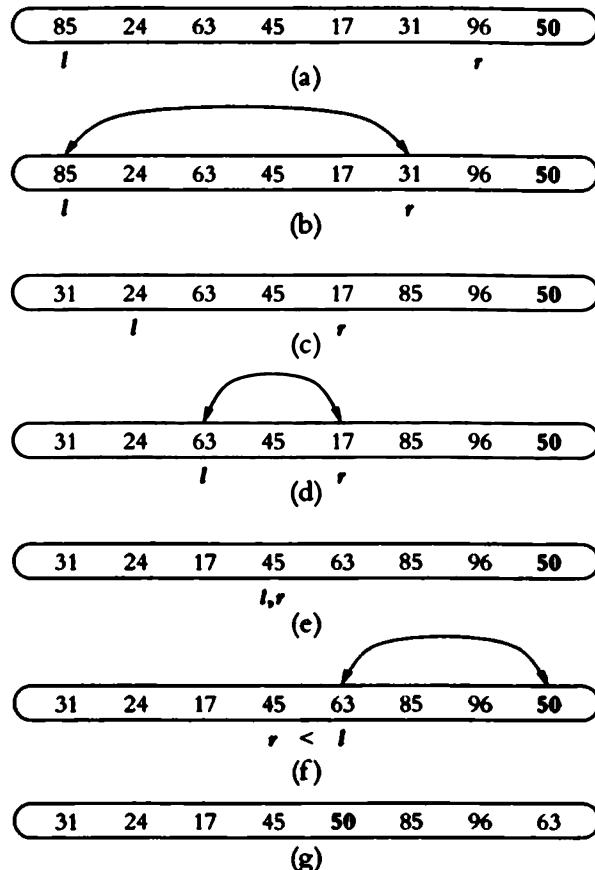


Figura 12.14: Fase di divisione nel quick-sort sul posto, usando l'indice l come abbreviazione per la variabile left e l'indice r per right. L'indice l scandisce la sequenza da sinistra a destra, mentre l'indice r lo fa da destra a sinistra. Viene eseguito uno scambio ogni volta che l corrisponde a un elemento non minore del pivot e r corrisponde a un elemento non maggiore del pivot. In (f) si può vedere lo scambio finale che porta il pivot nella sua posizione corretta e conclude la fase di divisione.

Selezione del pivot

L'implementazione presentata in questo paragrafo sceglie alla cieca l'ultimo elemento come pivot, a ogni livello della ricorsione di quick-sort. Questa strategia lascia aperta la porta a prestazioni temporali $\Theta(n^2)$ nel caso peggiore, in particolare quando la sequenza originaria è già ordinata, oppure è ordinata in senso inverso, oppure è "quasi" ordinata.

Come detto nel Paragrafo 12.2.1, questa situazione può essere migliorata usando una scelta casuale del pivot in ogni fase di suddivisione. Dal punto di vista pratico, un'altra tecnica molto usata per scegliere il pivot è quella di prendere il valore mediano tra tre valori, presi, rispettivamente, dalla posizione iniziale, centrale e finale dell'array. Questa strategia euristica, che prende il nome di *mediana di tre* valori, sceglierà più spesso un valore "buono" per il pivot e il calcolo del valore mediano tra tre valori può essere più veloce di una scelta casuale del pivot guidata da un generatore di numeri casuali. Per insiemi di dati di grandi dimensioni, si può anche calcolare il valore mediano di più di tre valori.

Approcci ibridi

Anche se quick-sort ha prestazioni davvero molto buone per insiemi di dati di grandi dimensioni, se la sequenza da ordinare è molto piccola questo algoritmo non è particolarmente efficiente, per via dei “costi fissi” (in termini di tempo). Ad esempio, la procedura per ordinare con quick-sort una sequenza di otto elementi, come illustrato nelle Figura 12.10, 12.11 e 12.12, richiede molta attività di gestione. In pratica, un algoritmo semplice, come l’ordinamento per inserimento (*insertion-sort*, visto nel Paragrafo 7.6) verrà eseguito più velocemente quando la sequenza da ordinare è così corta.

Quindi, nelle implementazioni dell’ordinamento particolarmente ottimizzate, è pratica diffusa adottare un approccio ibrido, usando un algoritmo dividi-e-conquista fino a quando la dimensione della sotto-sequenza in esame non scende al di sotto di una determinata soglia (ad esempio, 50 elementi), per invocare, poi, direttamente l’algoritmo di ordinamento per inserimento per le sotto-sequenze aventi dimensione inferiore alla soglia. Nel Paragrafo 12.4 discuteremo ulteriormente queste considerazioni pratiche, nell’ambito del confronto tra le prestazioni dei vari algoritmi di ordinamento.

12.3 Approfondimento algoritmico per lo studio dell’ordinamento

Riassumendo, fino a questo punto, la nostra trattazione dell’ordinamento, abbiamo descritto diversi metodi che hanno un tempo d’esecuzione, atteso o nel caso peggiore, $O(n \log n)$ per una sequenza da ordinare di dimensione n . Questi metodi comprendono merge-sort e quick-sort, descritti in questo capitolo, ma anche heap-sort, descritto nel Paragrafo 9.4.2. In questo paragrafo studieremo l’ordinamento come problema algoritmico, discutendo problemi generali che riguardano gli algoritmi di ordinamento.

12.3.1 Limite inferiore asintotico per l’ordinamento

Una domanda che viene naturale porsi a questo punto è se si possa ordinare in un tempo inferiore a $O(n \log n)$. Curiosamente, se l’operazione elementare usata da un algoritmo di ordinamento è il confronto tra due elementi, queste prestazioni sono, in effetti, le migliori che si possono ottenere: l’ordinamento basato su confronti ha un limite inferiore asintotico $\Omega(n \log n)$ per il suo tempo d’esecuzione nel caso peggiore (si ricordi la notazione $\Omega(\cdot)$ presentata nel Paragrafo 4.3.1). Ci concentreremo qui sul costo principale dell’ordinamento basato su confronti, contando soltanto i confronti, tanto cerchiamo un limite inferiore al tempo d’esecuzione.

Supponiamo di dover ordinare una sequenza $S = (x_0, x_1, \dots, x_{n-1})$ e ipotizziamo che tutti gli elementi di S siano distinti (questa ipotesi non è in realtà un vincolo restrittivo, dato che vogliamo ricavare un limite inferiore). Al fine di trovare un limite inferiore, non ci preoccupiamo del fatto che S sia realizzata con un array o con una lista concatenata, perché vogliamo contare solamente i confronti tra elementi. Ogni volta che un algoritmo di ordinamento confronta due elementi x_i e x_j (cioè si pone la domanda “ x_i è minore di x_j ?”), le risposte possibili sono due: “sì” o “no”. Sulla base dei risultati di un confronto, l’algoritmo di ordinamento esegue eventualmente qualche elaborazione (il cui costo qui trascuriamo) e, prima o poi, esegue un altro confronto tra due altri elementi di S , ottenendo di nuovo

una risposta tra due possibili. Un algoritmo di ordinamento basato su confronti può, quindi, essere rappresentato con un albero di decisione T (visto nell'Esempio 8.5), nel quale ogni nodo interno v corrisponde a un confronto e ogni ramo che porta dalla posizione v a uno dei suoi figli corrisponde all'elaborazione svolta dall'algoritmo in conseguenza della risposta "sì" o "no" che ha ottenuto dal confronto. È importante notare che l'ipotetico algoritmo di ordinamento in questione non ha probabilmente alcuna consapevolezza esplicita dell'albero T : questo rappresenta semplicemente tutte le possibili sequenze di confronti che un algoritmo di ordinamento può fare, partendo dal primo confronto (associato alla radice dell'albero) e terminando con l'ultimo confronto (associato al genitore di un nodo esterno).

Ogni possibile disposizione iniziale (o *permutazione*) degli elementi di S farà eseguire all'ipotetico algoritmo di ordinamento una specifica sequenza di confronti, lungo un percorso in T che scende dalla radice a un nodo esterno. Associamo, quindi, a ciascun nodo esterno v di T l'insieme delle permutazioni degli elementi di S che fanno terminare proprio in v l'esecuzione dell'algoritmo di ordinamento. L'osservazione più importante nella nostra discussione sul limite inferiore è che ogni nodo esterno v in T può rappresentare la sequenza di confronti eseguita dall'algoritmo per un numero di permutazioni di S al massimo uguale a uno. La dimostrazione di questa affermazione è semplice: se due permutazioni diverse P_1 e P_2 di S fossero associate allo stesso nodo esterno, esisterebbero almeno due elementi, x_i e x_j , tali che x_i precede x_j in P_1 ma x_i segue x_j in P_2 . Allo stesso tempo, il risultato prodotto dall'algoritmo di ordinamento quando giunge in v deve essere uno specifico ordinamento di S , con uno degli elementi, x_i o x_j , che precede l'altro. Ma, se tanto P_1 quanto P_2 fanno sì che l'algoritmo disponga gli elementi di S in questo stesso ordine, allora significa che esiste un modo per costringere l'algoritmo a disporre x_i e x_j nell'ordine relativo sbagliato. Dato che questo non è consentito a un algoritmo di ordinamento corretto, ogni nodo esterno di T deve essere associato a una e soltanto una permutazione di S . Usiamo questa proprietà dell'albero di decisione associato a un algoritmo di ordinamento per dimostrare il seguente risultato:

Proposizione 12.4: *Il tempo d'esecuzione di qualunque algoritmo di ordinamento basato su confronti applicato a una sequenza di dimensione n è $\Omega(n \log n)$ nel caso peggiore.*

Dimostrazione: Il tempo d'esecuzione di un algoritmo di ordinamento basato su confronti non può essere inferiore all'altezza dell'albero di decisione T associato a tale algoritmo, come già detto (si veda la Figura 12.15). Per la discussione appena riportata, ogni nodo esterno di T deve essere associato a una permutazione di S . Inoltre, ogni permutazione di S deve far giungere l'algoritmo a un diverso nodo esterno di T . Ricordando che il numero di permutazioni di n oggetti è $n! = n(n - 1)(n - 2)\dots 2 \cdot 1$, l'albero T deve avere almeno $n!$ nodi esterni. Per la Proposizione 8.7, l'altezza di T è, quindi, almeno $\log(n!)$. Questo dimostra immediatamente la proposizione, perché nel prodotto che definisce $n!$ esistono almeno $n/2$ fattori che sono non minori di $n/2$, quindi:

$$\log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \log \frac{n}{2}$$

che è $\Omega(n \log n)$. ■

Altezza minima
(cioè caso peggiore per il tempo d'esecuzione)

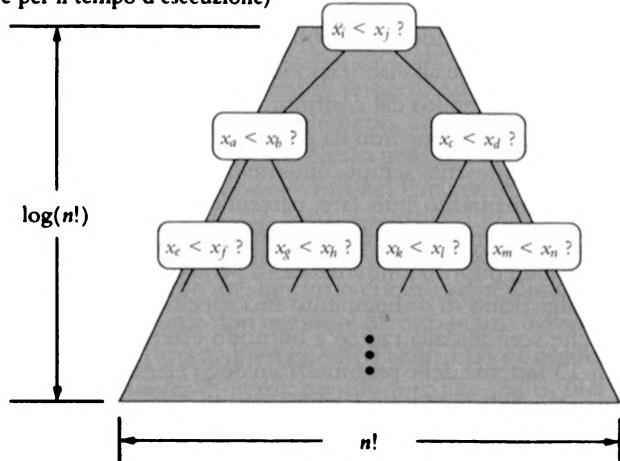


Figura 12.15: Analisi grafica del limite inferiore per gli algoritmi di ordinamento basati su confronti.

12.3.2 Ordinamento in tempo lineare: *bucket-sort* e *radix-sort*

Nel paragrafo precedente abbiamo dimostrato che, nel caso peggiore, per ordinare una sequenza di n elementi usando un algoritmo di ordinamento basato su confronti è necessario un tempo $\Omega(n \log n)$. La domanda che dovrebbe ora sorgere spontanea è se esistano altre categorie di algoritmi di ordinamento, che si possano progettare in modo che vengano eseguiti in un tempo asintoticamente inferiore a $\Omega(n \log n)$. Curiosamente, questi algoritmi esistono, ma necessitano di speciali ipotesi in merito alle sequenze da ordinare. Ciò nonostante, si tratta di situazioni che, nella pratica, si verificano spesso, ad esempio quando si ordinano numeri interi appartenenti a un intervallo noto oppure si ordinano stringhe di caratteri: per questo motivo è utile parlarne. In questo paragrafo considereremo il problema di ordinare una sequenza di voci, ciascuna delle quali è una coppia chiave-valore, dove le chiavi hanno dei vincoli di appartenenza a un determinato tipo di dato.

Bucket-sort

Consideriamo una sequenza S di n voci, le cui chiavi siano numeri interi appartenenti all'intervallo $[0, N - 1]$, con N intero e $N \geq 2$, che debba essere ordinata in base ai valori delle chiavi. In questo caso è possibile ordinare S in un tempo $O(n + N)$. Potrebbe sembrare un risultato inatteso: questo implica, ad esempio, che se N è $O(n)$, allora possiamo ordinare S in un tempo $O(n)$. Ovviamente, il punto critico è che, per via delle ipotesi restrittive fatte sugli elementi, si possa evitare di usare confronti.

L'idea si basa sull'utilizzo di un algoritmo, chiamato *bucket-sort*, che non è basato sui confronti tra elementi, ma sull'uso delle chiavi come indici all'interno di un array di bucket B le cui celle (o, appunto, *bucket*) abbiano indici che vanno da 0 a $N - 1$. Una voce che abbia chiave k viene posta nel bucket $B[k]$, che, a sua volta, è una sequenza (di voci aventi chiave k). Dopo aver inserito ogni voce della sequenza S nel bucket che le compete, possiamo

rimettere tutte le voci in S ordinatamente, scandendo il contenuto dei bucket $B[0], B[1], \dots, B[N - 1]$ in ordine. Nel Codice 12.7 descriviamo l'algoritmo bucket-sort.

Codice 12.7: L'algoritmo bucket-sort.

Algoritmo `bucketSort(S)`:

Input: Una sequenza S di voci con chiavi intere nell'intervallo $[0, N - 1]$

Output: La sequenza S con le voci ordinate secondo chiavi non decrescenti
sia B un array di n sequenze (dette *bucket*), ciascuna inizialmente vuota

for ogni voce e in S **do**

sia k la chiave di e

elimina e da S e inseriscila alla fine del bucket $B[k]$

for i che va da 0 a $n - 1$ **do**

for ogni voce e nel bucket $B[i]$ **do**

elimina e da $B[i]$ e inseriscila alla fine di S

È facile capire che l'algoritmo bucket-sort viene eseguito in un tempo $O(n + N)$ e usa una quantità $O(n + N)$ di spazio in memoria. Quindi, bucket-sort è efficiente quando l'intervallo N di valori delle chiavi è piccolo in confronto alla dimensione n della sequenza, diciamo $N = O(n)$ oppure $N = O(n \log n)$. Ancora, osserviamo che le prestazioni peggiorano se N aumenta di molto rispetto a n .

Come importante proprietà dell'algoritmo bucket-sort, citiamo il fatto che funziona correttamente anche se ci sono molti elementi diversi che hanno la stessa chiave. In effetti, l'abbiamo descritto in un modo che tiene già conto di questa eventualità.

Ordinamento stabile

Quando si ordinano coppie chiave-valore, è importante sapere come vengono gestite le chiavi uguali. Sia $S = ((k_0, v_0), \dots, (k_{n-1}, v_{n-1}))$ una sequenza di tali voci: diciamo che un algoritmo di ordinamento è *stabile (stable)* se, per qualsiasi coppia di voci di S , (k_i, v_i) e (k_j, v_j) , tali che $k_i = k_j$ e che (k_i, v_i) precede (k_j, v_j) in S prima dell'ordinamento (cioè, $i < j$), si ha che dopo l'ordinamento la voce (k_i, v_i) precede ancora la voce (k_j, v_j) . La stabilità è una proprietà importante, per un algoritmo di ordinamento, perché alcune applicazioni vogliono preservare l'ordinamento iniziale tra gli elementi aventi la stessa chiave.

La nostra descrizione informale di bucket-sort, proposta nel Codice 12.7, garantisce la stabilità a condizione che chi implementa l'algoritmo faccia in modo che tutte le sequenze si comportino come code, con elementi ispezionati e rimossi all'inizio della sequenza e inseriti alla fine. Detto altrimenti, quando all'inizio si introducono gli elementi di S nei bucket, si deve elaborare S dall'inizio verso la fine e ogni elemento va aggiunto alla fine del bucket a cui deve appartenere. Successivamente, quando si trasferiscono gli elementi dai bucket a S , si deve elaborare ciascun bucket dall'inizio verso la fine, aggiungendo i suoi elementi alla fine di S .

Radix-sort

Uno dei motivi per cui l'ordinamento stabile è così importante risiede nel fatto che consente di applicare l'approccio di bucket-sort anche a un contesto più generale di quello relativo all'ordinamento di numeri interi. Supponiamo, ad esempio, di voler ordinare voci le cui chiavi sono coppie (k, l) , dove k e l sono numeri interi appartenenti all'intervallo

$[0, N - 1]$, con N intero e $N \geq 2$. In un tale contesto, è comune definire un ordinamento tra queste chiavi usando la convenzione *lessicografica*, analoga a quella usata nei dizionari, dove $(k_1, l_1) < (k_2, l_2)$ se $k_1 < k_2$ oppure se $k_1 = k_2$ e $l_1 < l_2$. Si tratta di una versione “per coppie” della funzione di confronto lessicografico che può essere applicata a stringhe di uguale lunghezza o a tuple di lunghezza d .

L’algoritmo *radix-sort* (*ordinamento mediante radice*) ordina una sequenza S di voci, le cui chiavi sono coppie, applicando due volte l’ordinamento bucket-sort nella sua forma stabile, una prima volta usando come chiave di ordinamento una delle due componenti della coppia, poi usando l’altra componente. Ma in che ordine vanno usate le due componenti? Dobbiamo ordinare prima usando k (cioè la prima componente della coppia) e poi l (la seconda componente), oppure viceversa?

Per capire come si possa rispondere a questa domanda, consideriamo l’esempio seguente.

Esempio 12.5: Consideriamo la seguente sequenza S (di cui mostriamo soltanto le chiavi, che sono coppie):

$$S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))$$

Se ordiniamo S in modo stabile usando la prima componente delle coppie, otteniamo la sequenza:

$$S_1 = ((1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2))$$

Se, poi, ordiniamo S_1 in modo stabile usando la seconda componente, otteniamo la sequenza:

$$S_{1,2} = ((1, 2), (2, 2), (3, 2), (2, 3), (3, 3), (1, 5), (2, 5), (1, 7))$$

che, purtroppo, non è una sequenza ordinata. D’altra parte, se per prima cosa ordiniamo S in modo stabile usando la seconda componente della coppia, otteniamo la sequenza:

$$S_2 = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7))$$

Ordinando, poi, S_2 in modo stabile usando la prima componente, otteniamo la sequenza:

$$S_{2,1} = ((1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3))$$

che, in effetti, è proprio la sequenza S ordinata lessicograficamente.

Quindi, osservando questo esempio, siamo portati a credere che si debba per prima cosa ordinare usando la seconda componente, poi ordinare di nuovo usando la prima componente. Questa intuizione è corretta. Ordinando in modo stabile usando la seconda componente e, poi, ordinando di nuovo usando la prima componente, garantiamo che, se due voci risultano essere uguali durante la seconda procedura di ordinamento (cioè hanno la prima componente uguale), allora viene preservato il loro posizionamento relativo così come si presentava nella sequenza iniziale, prima dell’ordinamento avvenuto in base alla seconda componente. Quindi, la sequenza risultante è sempre lessicograficamente ordinata. Lasciamo

come semplice esercizio (R-12.19) l'estensione di questo approccio a triplete e, in generale, d -tuple di numeri. Possiamo, quindi, così riassumere questo paragrafo.

Proposizione 12.6: *Sia S una sequenza di n coppie chiave-valore, ciascuna delle quali contiene una chiave (k_1, k_2, \dots, k_d) , con k_i numero intero appartenente, per ogni i , all'intervallo $[0, N - 1]$, con N intero e $N \geq 2$. Usando radix-sort, possiamo ordinare lessicograficamente S in un tempo $O(d(n + N))$.*

L'algoritmo radix-sort può essere applicato a qualunque chiave che possa essere vista come un oggetto composto da componenti più piccole e ordinabile in senso lessicografico. Ad esempio, possiamo applicarlo per ordinare stringhe di caratteri di lunghezza contenuta, dal momento che ogni singolo carattere può essere visto come un numero intero (anche se bisogna gestire con attenzione l'ordinamento di stringhe di lunghezze diverse).

12.4 Confronto tra algoritmi di ordinamento

A questo punto potrebbe essere utile fermarci un momento e riprendere in esame tutti gli algoritmi di ordinamento che abbiamo studiato finora, considerando la loro applicazione a una sequenza di n elementi.

Tempo d'esecuzione e altri fattori

Abbiamo studiato diversi algoritmi, tra i quali l'ordinamento per inserimento e l'ordinamento per selezione, che hanno un comportamento temporale quadratico (cioè $O(n^2)$) tanto nel caso medio quanto nel caso peggiore. Abbiamo, inoltre, analizzato altri metodi che hanno un comportamento $O(n \log n)$, come heap-sort, merge-sort e quick-sort. Infine, abbiamo visto i metodi bucket-sort e radix-sort, che, per particolari tipi di chiavi, vengono eseguiti in un tempo lineare. L'algoritmo di ordinamento per selezione è certamente una scelta molto debole in qualsiasi applicazione, dal momento che richiede un tempo quadratico anche nel caso migliore, ma, tra gli altri algoritmi di ordinamento, qual è il migliore?

Come per molti altri problemi che si presentano nella vita, tra i candidati rimasti in corsa non c'è un algoritmo di ordinamento che sia "il migliore" in modo evidente e netto: occorre studiare compromessi che riguardano l'efficienza, l'utilizzo della memoria e la stabilità dell'ordinamento. La scelta dell'algoritmo di ordinamento migliore per una determinata applicazione dipende dalle proprietà dell'applicazione stessa e, proprio per questo motivo, l'algoritmo di ordinamento "standard" usato nei linguaggi di programmazione e nei sistemi di calcolo ha subito molte modifiche nel tempo. Possiamo, però, dare qui qualche suggerimento, basandoci sulle proprietà note dei diversi algoritmi di ordinamento "validi", cioè meritevoli di attenzione.

Insertion-sort

Il tempo d'esecuzione dell'algoritmo di ordinamento per inserimento (*insertion-sort*), se ben implementato, è $O(n + m)$, dove m è il numero di *inversioni* presenti nella sequenza da ordinare, cioè il numero di coppie di elementi che sono fuori ordine. Quindi, insertion-sort è un algoritmo eccellente per ordinare sequenze di piccole dimensioni (diciamo fino a 50 elementi), perché è un algoritmo semplice da implementare e le sequenze piccole hanno

necessariamente poche inversioni. Ancora, l'algoritmo insertion-sort è piuttosto efficiente nell'ordinamento di sequenze che siano già “quasi” ordinate, intendendo per “quasi ordinate” il fatto che il numero di inversioni presenti sia piccolo. Le prestazioni quadratiche dell'algoritmo di ordinamento per inserimento, però, lo rendono una scelta veramente debole al di fuori di questi contesti molto particolari.

Heap-sort

L'algoritmo **heap-sort**, d'altro canto, viene eseguito in un tempo $O(n \log n)$ anche nel caso peggiore, una prestazione ottima per i metodi di ordinamento basati su confronti. L'algoritmo heap-sort può essere facilmente modificato per l'esecuzione sul posto e costituisce una scelta naturale per sequenze di piccole e medie dimensioni, quando i dati da elaborare trovano posto nella memoria principale, anche se per sequenze più lunghe tende a essere sconfitto, nel tempo d'esecuzione, tanto da quick-sort quanto da merge-sort. Per via degli scambi tra elementi, l'implementazione standard di heap-sort non effettua un ordinamento stabile.

Quick-sort

Nonostante le prestazioni quadratiche nel caso peggiore rendano l'algoritmo **quick-sort** poco adatto in applicazioni in tempo reale (*real-time*), dove occorre fornire garanzie in merito al tempo richiesto per portare a termine un'operazione di ordinamento, le sue prestazioni attese sono $O(n \log n)$ e diversi studi sperimentali hanno evidenziato che è spesso in grado di battere tanto heap-sort quanto merge-sort. Per via degli scambi tra elementi che avvengono durante la fase di divisione, l'algoritmo quick-sort non è in grado di garantire la stabilità dell'ordinamento in modo naturale.

Per decenni quick-sort ha rappresentato la scelta standard come algoritmo di ordinamento generico, operante all'interno della memoria principale: l'algoritmo quick-sort era implementato nel metodo di ordinamento `qsort` messo a disposizione dalle librerie del linguaggio C e per molti anni è stato alla base dei metodi di ordinamento nei sistemi operativi Unix. Anche nella libreria standard di Java, è stato da sempre scelto come algoritmo di ordinamento per array di tipi primitivi (tra poco parleremo, invece, dell'ordinamento di oggetti).

Merge-sort

L'algoritmo **merge-sort** viene eseguito in un tempo $O(n \log n)$ nel caso peggiore. È abbastanza difficile fare in modo che merge-sort operi sul posto per ordinare un array e, senza tale ottimizzazione, i costi aggiuntivi necessari per l'utilizzo di un array temporaneo e per copiare i dati da un array all'altro lo rendono meno interessante delle implementazioni di heap-sort e quick-sort che operano sul posto, in tutti quei casi in cui la sequenza da ordinare trova posto completamente nella memoria principale del calcolatore. Nonostante questo, merge-sort è un algoritmo eccellente per quelle situazioni in cui i dati sono disposti all'interno di vari livelli gerarchici della memoria del calcolatore (ad esempio, memoria *cache*, memoria principale, memoria esterna). In tali contesti, il fatto che merge-sort elabori lunghe sequenze di dati sotto forma di flussi da fondere insieme consente di utilizzare nel modo migliore i dati trasferiti a blocchi da un livello all'altro della memoria, riducendo così il numero complessivo di trasferimenti di blocchi all'interno della gerarchia delle memorie.

Il metodo di ordinamento del pacchetto GNU e i metodi di ordinamento usati dalle varie versioni del sistema operativo Linux si basano su una variante di merge-sort a più vie (*multiway merge-sort*). L'algoritmo **Tim-sort** (progettato da Tim Peters) è un approccio ibrido

che, in estrema sintesi, utilizza una variante *bottom-up* di merge-sort che sfrutta la presenza di sotto-sequenze ordinate nella sequenza iniziale, usando insertion-sort per la costruzione di ulteriori sotto-sequenze ordinate. Tim-sort è l'algoritmo d'ordinamento standard del linguaggio Python fin dal 2003 e lo è diventato anche in Java SE 7 per l'ordinamento di array di oggetti.

Bucket-sort e Radix-sort

Infine, se un'applicazione prevede l'ordinamento di voci aventi come chiavi numeri interi di piccolo valore, oppure stringhe di caratteri o, ancora, d -tuple di chiavi appartenenti a un insieme discreto, gli algoritmi *bucket-sort* o *radix-sort* rappresentano una scelta eccellente, perché vengono eseguiti in un tempo $O(d(n + N))$, dove $[0, N - 1]$ è l'intervallo a cui appartengono le chiavi intere e $d = 1$ per bucket-sort. Quindi, se $d(n + N)$ è significativamente "inferiore" alla funzione $n \log n$, allora questo metodo di ordinamento viene generalmente eseguito più velocemente di quick-sort, heap-sort e merge-sort.

12.5 Selezione

Nonostante sia così importante, l'ordinamento non è il solo problema interessante che riguarda una relazione d'ordine totale tra gli elementi di un insieme. Ci sono molte applicazioni che sono interessate a individuare un singolo elemento di un insieme in funzione della sua posizione relativa all'interno della disposizione ordinata di tutti gli elementi dell'insieme stesso. Tra gli esempi di questo, citiamo l'individuazione dell'elemento minimo o dell'elemento massimo, ma anche, diciamo, l'elemento *mediano*, cioè tale che metà degli elementi dell'insieme sono minori e l'altra metà sono maggiori dell'elemento in oggetto. In generale, i problemi che chiedono di conoscere un elemento in base alla sua posizione all'interno di una disposizione ordinata sono dette *statistiche di ordine*.

Definizione del problema della selezione

In questo paragrafo parleremo del generico problema, nell'ambito delle statistiche di ordine, che riguarda l'individuazione (o *selezione*, *selection*) del k -esimo elemento più piccolo all'interno di un contenitore non ordinato di n elementi confrontabili: si tratta, appunto, del *problema della selezione*. Possiamo ovviamente risolvere questo problema ordinando il contenitore e, poi, accedendo all'elemento aente indice $k - 1$ nella sequenza ordinata. Usando il migliore tra gli algoritmi di ordinamento basati su confronti, questo approccio richiederebbe un tempo $O(n \log n)$, che è ovviamente eccessivo nei casi in cui $k = 1$ o $k = n$ (o anche $k = 2$, $k = 3$, $k = n - 1$ o anche $k = n - 5$), perché per tali valori di k possiamo risolvere facilmente il problema della selezione in un tempo $O(n)$. Quindi, viene naturale chiedersi se si possa risolvere il problema della selezione in un tempo $O(n)$ per qualsiasi valore di k e, in particolare, per il caso interessante del valore mediano, dove $k = \lfloor n/2 \rfloor$.

12.5.1 Prune-and-Search (riduzione-e-ricerca)

Effettivamente è possibile risolvere il problema della selezione in un tempo $O(n)$ per qualsiasi valore di k e la tecnica che usiamo qui per ottenere tale risultato è un esempio

di un interessante schema di progettazione di algoritmi, che prende il nome di *prune-and-search* (“riduzione o potatura e ricerca”) o *decrease-and-conquer* (cioè “riduci di dimensione e conquista”). Applicando questo schema di progettazione, risolviamo un problema definito per un contenitore di n oggetti eliminando alcuni di tali n oggetti e risolvendo ricorsivamente il problema di dimensione inferiore. Quando, prima o poi, arriviamo a dover risolvere un problema ridotto a un contenitore di oggetti di dimensione costante, possiamo farlo con qualche metodo che usa la “forza bruta” (*brute-force method*); risalendo, poi, a ritroso lungo la pila delle invocazioni ricorsive, si completa la costruzione della soluzione del problema originario. In alcuni casi si può evitare l’uso della ricorsione, eseguendo semplicemente un ciclo di fasi di “potatura” dello schema *prune-and-search* fino a poter applicare il metodo a forza bruta. Incidentalmente, il metodo di ricerca binaria (o per bisezione) descritto nel Paragrafo 5.1.3 è un esempio dello schema di progettazione *prune-and-search*.

12.5.2 Quick-select probabilistico

Come applicazione dello schema *prune-and-search* alla ricerca del k -esimo elemento più piccolo in una sequenza non ordinata di n elementi, descriviamo un algoritmo semplice e pratico, che prende il nome di *randomized quick-select* o *quick-select probabilistico*. Questo algoritmo ha un tempo d’esecuzione atteso $O(n)$, facendo la media su tutte le possibili scelte casuali effettuate dall’algoritmo stesso; questo risultato atteso non dipende da alcuna ipotesi di casualità a cui debbano rispondere i dati elaborati. Osserviamo, però, che il quick-select probabilistico richiede, nel caso peggiore, un tempo d’esecuzione $O(n^2)$ e la dimostrazione di questa caratteristica è lasciata come esercizio (R-12.25). Presentiamo anche un esercizio (C-12.56) che modifica l’algoritmo quick-select probabilistico definendone una variante *deterministica* che viene eseguita in un tempo $O(n)$ nel caso peggiore: tuttavia, l’esistenza di questo algoritmo deterministico è principalmente di interesse teorico, per via del fattore costante “nascosto” dalla notazione O-grande, che in questo caso ha un valore relativamente elevato.

Supponiamo di disporre di una sequenza non ordinata S contenente n elementi confrontabili tra loro e di un numero intero $k \in [1, n]$. Ragionando a un elevato livello di astrazione, l’algoritmo quick-select per trovare il k -esimo elemento più piccolo in S è simile all’algoritmo quick-sort probabilistico descritto nel Paragrafo 12.2.1. Scegliamo casualmente un elemento “pivot” in S e lo usiamo per suddividere la sequenza S in tre sotto-sequenze, L , E e G , contenenti, rispettivamente, gli elementi di S che sono minori del pivot, uguali al pivot e maggiori del pivot. Durante la fase “prune” (cioè di “potatura”), determiniamo quale di questi sottoinsiemi di S contiene l’elemento che stiamo cercando, sulla base del valore di k e delle dimensioni dei sottoinsiemi stessi. Poi, eseguiamo una ricorsione agente sul solo sottoinsieme individuato, osservando che, in tale sottoinsieme, il rango dell’elemento cercato, che è uguale a k in S , può essere diverso. Il Codice 12.8 presenta l’algoritmo quick-select probabilistico, descritto mediante pseudocodice.

Codice 12.8: L’algoritmo quick-select probabilistico.

Algoritmo `quickSelect(S, k)`:

Input: Una sequenza S di n elementi confrontabili e un numero intero $k \in [1, n]$

Output: Il k -esimo elemento più piccolo in S

if $n == 1$ **then**

return il primo e unico elemento di S

scegli a caso un elemento x di S (il pivot) e dividi S in tre sequenze

- L , contenente gli elementi di S che sono minori di x
- E , contenente gli elementi di S che sono uguali a x
- G , contenente gli elementi di S che sono maggiori di x

if $k \leq |L|$ **then**

return quickSelect(L , k)

else if $k \leq |L| + |E|$ **then**

return x

{ ogni elemento di E è uguale a x }

else

return quickSelect(G , $k - |L| - |E|$) { notare il nuovo parametro di selezione }

12.5.3 Analisi del quick-select probabilistico

Per dimostrare che l'algoritmo quick-select probabilistico viene eseguito in un tempo atteso $O(n)$ servono, ovviamente, semplici basi di teoria della probabilità. La dimostrazione si basa sulla proprietà di *linearità del valore atteso*, secondo la quale, se X e Y sono variabili casuali e c è un numero, allora:

$$E(X + Y) = E(X) + E(Y) \quad \text{e} \quad E(cX) = cE(Y)$$

dove abbiamo usato $E(Z)$ per indicare il valore atteso dell'espressione Z .

Indichiamo con $t(n)$ il tempo d'esecuzione dell'algoritmo quick-select probabilistico applicato a una sequenza di dimensione n . Dato che questo algoritmo dipende da eventi casuali, il suo tempo d'esecuzione, $t(n)$, è una variabile casuale. Vogliamo, quindi, trovare un limite superiore per $E(t(n))$, che è il valore atteso di $t(n)$. Diciamo che un'invocazione ricorsiva del nostro algoritmo è "buona" se suddivide la sequenza che deve elaborare in modo che la dimensione di ciascuna delle due sotto-sequenze, L e G , sia al massimo $3n/4$. Chiaramente un'invocazione ricorsiva è buona con probabilità almeno $1/2$. Indichiamo con $g(n)$ il numero di invocazioni ricorsive consecutive che vengono effettuate, compresa quella in esame, prima di trovarne una buona. A questo punto, possiamo caratterizzare $t(n)$ usando la seguente *equazione di ricorrenza*:

$$t(n) \leq bn \cdot g(n) + t(3n/4),$$

dove $b \geq 1$ è una costante. Applicando la linearità del valore atteso, per $n > 1$ otteniamo:

$$E(t(n)) \leq E(bn \cdot g(n) + t(3n/4)) = bn \cdot E(g(n)) + Et(3n/4)).$$

Dato che un'invocazione ricorsiva è buona con probabilità almeno $1/2$, e il fatto che un'invocazione ricorsiva sia buona oppure no è un evento indipendente dal fatto che l'invocazione che l'ha invocata fosse buona, il valore atteso di $g(n)$ è al massimo uguale al numero di volte che ci aspettiamo di dover lanciare una moneta che si comporta equamente prima che esca "testa": quindi, $E(g(n)) \leq 2$. Di conseguenza, se, per brevità, indichiamo con $T(n)$ il valore atteso $E(t(n))$, possiamo scrivere che, per $n > 1$, si ha:

$$T(n) \leq T(3n/4) + 2bn.$$

Per convertire questa relazione in una forma chiusa, applichiamo ripetutamente la diseguaglianza nell'ipotesi che n sia grande a sufficienza. Ad esempio, dopo due applicazioni otteniamo:

$$T(n) \leq T((3n/4)^2 n) + 2b(3/4)n + 2bn.$$

A questo punto, dovrebbe essere evidente che il caso generale è:

$$T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i$$

In altre parole, il tempo d'esecuzione atteso è al massimo $2bn$ volte il valore di una somma geometrica la cui base è un numero positivo minore di 1. Quindi, per la Proposizione 4.5, $T(n)$ è $O(n)$.

Proposizione 12.7: *Il tempo d'esecuzione atteso dell'algoritmo quick-select probabilistico applicato a una sequenza S di dimensione n è $O(n)$, nell'ipotesi che due qualsiasi elementi di S possano essere confrontati tra loro in un tempo $O(1)$.*

12.6 Esercizi

Riepilogo e approfondimento

- R-12.1 Dimostrare la Proposizione 12.1.
- R-12.2 Nell'albero merge-sort rappresentato nelle Figure 12.2, 12.3 e 12.4, alcuni rami sono stati disegnati sotto forma di freccia. Qual è il significato di una freccia rivolta verso il basso? E quello di una rivolta verso l'alto?
- R-12.3 Dimostrare che il tempo d'esecuzione dell'algoritmo merge-sort applicato a una sequenza di n elementi è $O(n \log n)$ anche quando n non è una potenza di 2.
- R-12.4 La nostra implementazione di merge-sort che opera su array, descritta nel Paragrafo 12.1.2, è stabile? Spiegare perché, oppure perché no.
- R-12.5 La nostra implementazione di merge-sort che opera su liste concatenate, descritta nel Codice 12.3, è stabile? Spiegare perché, oppure perché no.
- R-12.6 Un algoritmo che ordina voci di tipo chiave-valore sulla base della chiave è detto *irregolare (straggling)* se, ogni volta che due voci e_i e e_j hanno chiave uguale, ma e_i precede e_j nella sequenza iniziale, l'algoritmo posiziona le due voci, nella sequenza finale, in modo che e_i segua e_j . Descrivere come si possa modificare l'algoritmo merge-sort visto nel Paragrafo 12.1 in modo che sia "irregolare".
- R-12.7 Sono date due sequenze ordinate, A e B , contenenti ciascuna n elementi distinti, ma alcuni elementi potrebbero essere presenti in entrambe le sequenze. Descrivere un metodo che, in un tempo $O(n)$, generi una sequenza ordinata, priva di duplicati, che rappresenti l'unione $A \cup B$.

- R-12.8 Descrivere mediante pseudocodice i metodi `retainAll` e `removeAll` dell'ADT "insieme", nell'ipotesi che si usino sequenze ordinate per implementarlo.
- R-12.9 Supponiamo di modificare la versione deterministica dell'algoritmo quick-sort in modo che, invece di scegliere sempre come pivot l'ultimo elemento di una sequenza di lunghezza n , scelga sempre l'elemento corrispondente all'indice $\lfloor n/2 \rfloor$. Qual è il tempo d'esecuzione di questa versione di quick-sort se la sequenza da ordinare è, in realtà, già ordinata?
- R-12.10 Supponiamo di modificare la versione deterministica dell'algoritmo quick-sort come descritto nell'esercizio precedente. Descrivere il tipo di sequenza che porterebbe il tempo d'esecuzione a essere $\Theta(n^2)$.
- R-12.11 Nell'ipotesi che il metodo `quickSortInPlace` venga eseguito per una sequenza che contiene elementi duplicati, dimostrare che l'algoritmo ordina ancora correttamente la sequenza. Cosa succede nella fase di suddivisione quando ci sono elementi uguali al pivot? Qual è il tempo d'esecuzione dell'algoritmo se tutti gli elementi sono uguali?
- R-12.12 Dimostrare che il tempo d'esecuzione nel caso migliore di quick-sort applicato a una sequenza di n elementi distinti è $\Theta(n \log n)$.
- R-12.13 Se il ciclo `while` più esterno presente nella nostra implementazione di `quickSortInPlace` (riga 9 del Codice 12.6) fosse modificato in modo da usare la condizione `left < right` invece di `left <= right`, si commetterebbe un errore. Spiegare qual è l'errore e indicare quale sia una specifica sequenza che fa fallire una tale implementazione.
- R-12.14 Se, nell'implementazione del nostro metodo `quickSortInPlace`, descritta nel Codice 12.6, modificassimo l'enunciato condizionale della riga 14 in modo da usare la condizione `left < right` invece di `left <= right`, si commetterebbe un errore. Spiegare qual è l'errore e indicare quale sia una specifica sequenza che fa fallire una tale implementazione.
- R-12.15 Seguendo come traccia la nostra analisi dell'algoritmo quick-sort probabilistico, descritta nel Paragrafo 12.2.1, dimostrare che la probabilità che un dato elemento x appartenga a più di $2 \log n$ sotto-problemi nel gruppo di dimensione i è al massimo $1/n^2$.
- R-12.16 Tra le $n!$ possibili configurazioni di ingresso fornite a un algoritmo di ordinamento che operi mediante confronti, qual è, in assoluto, il massimo numero di configurazioni che possono essere ordinate in modo corretto usando soltanto n confronti?
- R-12.17 Jonathan dispone di un algoritmo di ordinamento basato su confronti che ordina i primi k elementi di una sequenza di dimensione n in un tempo $O(n)$. Caratterizzare, mediante O-grande, il valore massimo che può assumere k .
- R-12.18 L'algoritmo bucket-sort opera sul posto? Spiegare perché, oppure perché no.
- R-12.19 Descrivere un metodo basato sull'algoritmo radix-sort per ordinare in senso lessicografico una sequenza S di tripletti (k, l, m) , dove k, l e m sono numeri interi appartenenti all'intervallo $[0, N - 1]$, con N intero e $N \geq 2$. Come si può estendere questo schema alla gestione di d -tuple (k_1, k_2, \dots, k_d) , dove ciascun valore k_i , per i che va da 1 a d , è un numero intero che appartiene all'intervallo $[0, N - 1]$?
- R-12.20 Supponiamo che S sia una sequenza di n valori, ciascuno uguale a 0 o a 1. Quanto tempo servirà per ordinare S con l'algoritmo merge-sort? E con quick-sort?

- R-12.21 Supponiamo che S sia una sequenza di n valori, ciascuno uguale a 0 o a 1. Quanto tempo servirà per ordinare S in modo stabile con l'algoritmo bucket-sort?
- R-12.22 Supponiamo che S sia una sequenza di n valori, ciascuno uguale a 0 o a 1. Descrivere un metodo per ordinare S sul posto.
- R-12.23 Descrivere un esempio di sequenza che venga ordinata da merge-sort e heap-sort in un tempo $O(n \log n)$, ma possa essere ordinata da insertion-sort in un tempo $O(n)$. Cosa succede se si inverte il contenuto di tale sequenza?
- R-12.24 Giustificando le risposte date, indicare l'algoritmo migliore per ordinare ciascuno dei seguenti tipi di dati: generici oggetti confrontabili, lunghe stringhe di caratteri, numeri interi a 32 bit, numeri in virgola mobile in doppia precisione, byte.
- R-12.25 Dimostrare che, nel caso peggiore, il tempo d'esecuzione dell'algoritmo quick-select applicato a una sequenza di n elementi è $\Omega(n^2)$.

Creatività

- C-12.26 Descrivere e analizzare un metodo efficiente per la rimozione di tutti i duplicati da una collezione A contenente n elementi.
- C-12.27 Aggiungere alla classe `LinkedPositionalList`, vista nel Paragrafo 7.3, il supporto per un metodo, di nome `sort`, che ordini gli elementi della lista ricollegando i nodi esistenti: non deve creare nessun nuovo nodo. Si può scegliere liberamente un algoritmo di ordinamento qualsiasi.
- C-12.28 Linda afferma di aver progettato un algoritmo che riceve una sequenza S e genera una sequenza T ordinata e contenente gli stessi n elementi di S .
- Descrivere un algoritmo, `isSorted`, che verifichi in un tempo $O(n)$ che T sia ordinata.
 - Spiegare perché l'algoritmo `isSorted` non è sufficiente a dimostrare che una particolare sequenza T prodotta dall'algoritmo di Linda sia una versione ordinata della sequenza S .
 - Descrivere quali ulteriori informazioni potrebbero essere generate dall'algoritmo di Linda per fare in modo che si possa stabilire in un tempo $O(n)$ il suo corretto funzionamento, data una qualsiasi coppia di sequenze, S e T , quest'ultima generata dall'algoritmo di Linda elaborando S .
- C-12.29 Aggiungere alla classe `LinkedPositionalList`, vista nel Paragrafo 7.3, il supporto per un metodo, di nome `merge`, avente il seguente comportamento. Se A e B sono esemplari di `LinkedPositionalList` i cui elementi sono ordinati, la sintassi `A.merge(B)` deve fondere tutti gli elementi di B in A , in modo che A resti ordinata e B rimanga vuota. Il metodo deve realizzare la fusione ricollegando i nodi esistenti: non deve creare nessun nuovo nodo.
- C-12.30 Implementare una versione *bottom-up* di merge-sort che ordini un contenitore di oggetti trasferendo per prima cosa ciascun suo elemento in una propria coda, per poi fondere ripetutamente coppie di code finché tutti gli elementi non si trovino, ordinati, all'interno di un'unica coda.
- C-12.31 Modificare l'implementazione di quick-sort operante sul posto che abbiamo presentato nel Codice 12.6 in modo che diventi una versione *probabilistica* dell'algoritmo, in analogia con quanto visto nel Paragrafo 12.2.1.

C-12.32 Considerare una versione deterministica di quick-sort in cui si sceglie ogni volta come pivot il valore mediano tra gli ultimi d elementi della sequenza di n elementi che bisogna ordinare, con d numero fisso, costante, dispari e $d \geq 3$. Qual è il tempo d'esecuzione asintotico nel caso peggiore per questa versione di quick-sort?

C-12.33 Le prestazioni dell'algoritmo quick-sort probabilistico possono essere analizzate anche usando un'*equazione di ricorrenza*. In questo caso, indichiamo con $T(n)$ il tempo atteso per l'esecuzione dell'algoritmo quick-sort probabilistico e osserviamo che, per effetto del caso peggiore delle suddivisioni nei casi "buono" e "cattivo", possiamo scrivere:

$$T(n) \leq \frac{1}{2}(T(3n/4) + T(n/4)) + \frac{1}{2}(T(n-1)) + bn$$

dove bn è il tempo richiesto per suddividere una lista in base a un determinato pivot e, poi, concatenare le sotto-liste risultanti dopo che le invocazioni ricorsive sono terminate. Dimostrare, per induzione, che $T(n)$ è $O(n \log n)$.

C-12.34 La nostra descrizione ad alto livello di quick-sort prevede di partizionare gli elementi in tre insiemi, L , E e G , contenenti, rispettivamente, le chiavi che sono minori del pivot, quelle che sono uguali al pivot e quelle che sono maggiori del pivot. Tuttavia, la nostra implementazione di quick-sort che opera sul posto, nel Codice 12.6, non porta nell'insieme E tutti gli elementi uguali al pivot. Vediamo, quindi, una strategia alternativa per eseguire il partizionamento in tre insiemi operando sul posto. Si scandiscono con un ciclo tutti gli elementi, da sinistra a destra, usando gli indici a , b e c , e preservando la condizione invariante che afferma che gli elementi aventi indice i tale che $0 \leq i < a$ sono minori del pivot, quelli con indice i tale che $a \leq i < b$ sono uguali al pivot e quelli con indice i tale che $b \leq i < c$ sono maggiori del pivot; gli elementi con indice i tale che $c \leq i < n$ non sono ancora stati catalogati. In ciascuna iterazione del ciclo viene catalogato un nuovo elemento, eseguendo, se necessario, un numero costante di scambi. Implementare una versione di quick-sort operante sul posto che segua questa strategia.

C-12.35 Immaginiamo che sia disponibile una sequenza S contenente n elementi, ciascuno dei quali rappresenta un diverso voto espresso per le elezioni presidenziali, dove ciascun voto è un numero intero associato a un particolare candidato, anche se i numeri interi utilizzati possono essere arbitrariamente grandi (anche se il numero di candidati non lo è). Progettare un algoritmo che in un tempo $O(n \log n)$ decida chi ha vinto le elezioni rappresentate da S , nell'ipotesi che vinca semplicemente il candidato che ha ottenuto il maggior numero di voti.

C-12.36 In relazione al problema della votazione visto nell'Esercizio C-12.35, supponiamo ora di conoscere il numero $k < n$ di candidati in corsa per l'elezione, anche se, di nuovo, il numero intero ID associato a ciascun candidato può essere arbitrariamente grande. Progettare un algoritmo che in un tempo $O(n \log k)$ decida chi ha vinto le elezioni.

C-12.37 In relazione al problema della votazione visto nell'Esercizio C-12.35, supponiamo ora che siano stati usati i numeri interi da 1 a k per individuare i

- candidati, con $k < n$. Progettare un algoritmo che in un tempo $O(n)$ decida chi ha vinto le elezioni.
- C-12.38 Dimostrare che qualsiasi algoritmo di ordinamento basato su confronti può essere reso stabile senza penalizzare le sue prestazioni temporali asintotiche.
- C-12.39 Supponiamo che siano date due sequenze, A e B , contenenti ciascuna n elementi (con la possibile presenza di duplicati), per i quali sia definita una relazione d'ordine totale. Descrivere un algoritmo efficiente per determinare se A e B contengono lo stesso insieme di elementi. Qual è il tempo d'esecuzione dell'algoritmo progettato?
- C-12.40 Dato un array A di n numeri interi appartenenti all'intervallo $[0, n^2 - 1]$, descrivere un metodo semplice per ordinare A in un tempo $O(n)$.
- C-12.41 Siano S_1, S_2, \dots, S_k k sequenze distinte i cui elementi hanno come chiavi numeri interi appartenenti all'intervallo $[0, N - 1]$, con N intero e $N \geq 2$. Descrivere un algoritmo che generi le k sequenze ordinate corrispondenti in un tempo $O(n + N)$, dove n indica la somma delle dimensioni di tutte le k sequenze.
- C-12.42 Data una sequenza S di n elementi, tra i quali sia definita una relazione d'ordine totale, descrivere un metodo efficiente per determinare se in S esistono due elementi uguali. Qual è il tempo d'esecuzione del metodo progettato?
- C-12.43 Sia S una sequenza di n elementi, all'interno della quale è definita una relazione d'ordine totale. Ricordando che un'*inversione* in S è una coppia di elementi x e y tali che x precede y in S ma $x > y$, descrivere un algoritmo che in un tempo $O(n \log n)$ determini il *numero* di inversioni presenti in S .
- C-12.44 Sia S una sequenza di n numeri interi. Descrivere un metodo che in un tempo $O(n + k)$ visualizzi tutte le coppie in S che danno luogo a un'inversione, così come definita nell'esercizio precedente, essendo k il numero di tali inversioni.
- C-12.45 Sia S una permutazione casuale di n numeri interi distinti. Dimostrare che il tempo d'esecuzione atteso dell'algoritmo insertion-sort applicato a S è $\Omega(n^2)$ (*suggerimento*: osservare che ci si può aspettare che metà degli elementi che, nella versione ordinata di S , si trovano nella metà contenente gli elementi maggiori si trovino inizialmente nella prima metà di S).
- C-12.46 Siano A e B due sequenze, ciascuna delle quali contiene n numeri interi. Dato un numero intero m , descrivere un algoritmo che in un tempo $O(n \log n)$ determini se esistono due numeri interi, il primo, a , appartenente alla sequenza A e il secondo, b , appartenente a B , tali che $m = a + b$.
- C-12.47 Dati due insiemi A e B rappresentati mediante sequenze ordinate, descrivere un algoritmo efficiente per calcolare $A \oplus B$, che è l'insieme degli elementi che sono in A o in B , ma non in entrambi.
- C-12.48 Dato un insieme di n numeri interi, descrivere e analizzare un metodo veloce per trovare i k numeri interi più vicini al valore mediano, con $k = \lceil \log n \rceil$.
- C-12.49 Bob ha un insieme A di n dadi e un insieme B di n bulloni, tali che per ciascun dado di A esiste un unico bullone in B avente dimensione corrispondente. Sfortunatamente, tutti i dati di A sembrano uguali, così come tutti i bulloni di B . L'unica modalità di confronto disponibile per Bob consiste nel prendere una coppia dado-bullone (a, b) , con a in A e b in B , e verificare se la filettatura di a corrisponde a quella di b oppure è più grande o più piccola. Descrivere e analizzare un algoritmo efficiente che consenta a Bob di accoppiare tutti i suoi dadi e bulloni.

- C-12.50 La nostra implementazione di quick-select può essere resa più efficiente in quanto a spazio occupato in memoria calcolando inizialmente soltanto le *dimensioni* degli insiemi L , E e G , creando poi solo i sottoinsiemi che sono effettivamente necessari per la ricorsione. Implementare questa nuova versione.
- C-12.51 Descrivere mediante pseudocodice una versione dell'algoritmo quick-select che operi sul posto, ipotizzando che si possa modificare l'ordine degli elementi.
- C-12.52 Dimostrare come si possa ordinare una sequenza di n elementi in un tempo che sia $O(n \log n)$ nel caso peggiore usando l'algoritmo deterministico che risolve in un tempo $O(n)$ il problema della selezione.
- C-12.53 Data una sequenza non ordinata S contenente n elementi confrontabili tra loro e un numero intero k , descrivere un algoritmo che, in un tempo atteso $O(n \log k)$, trovi gli elementi, in numero $O(k)$, che hanno rango $\lceil n/k \rceil, 2\lceil n/k \rceil, 3\lceil n/k \rceil$, e così via.
- C-12.54 Gli alieni ci hanno fornito un metodo, `alienSplit`, che riceve una sequenza S contenente n numeri interi e in un tempo $O(n)$ la partiziona nelle sequenze S_1, S_2, \dots, S_k , ciascuna delle quali avente dimensione massima $\lceil n/k \rceil$, tali che gli elementi di S_i sono non maggiori di qualsiasi elemento di S_{i+1} , per $i = 1, 2, \dots, k-1$, con k costante e $k < n$. Descrivere come si possa usare il metodo `alienSplit` per ordinare S in un tempo $O(n (\log n) / (\log k))$.
- C-12.55 Seguendo la traccia presentata nei cinque punti di questo esercizio, dimostrare che l'algoritmo quick-sort probabilistico viene eseguito in un tempo $O(n \log n)$ con probabilità non inferiore a $1 - 1/n$, cioè con *elevata probabilità*.
- Per ogni elemento x della sequenza da ordinare, indichiamo con $C_{i,j}(x)$ una variabile casuale booleana che vale 1 se e solo se l'elemento x appartiene a $j + 1$ sotto-problemi che hanno una dimensione s tale che $(3/4)^{j+1} n < s \leq (3/4)^j n$, altrimenti vale 0. Spiegare perché non c'è bisogno di definire $C_{i,j}$ per $j > n$.
 - Indichiamo con $X_{i,j}$ una variabile casuale booleana indipendente che vale 1 con probabilità $1/2^j$ e poniamo $L = \lceil \log_{4/3} n \rceil$. Dimostrare che $\sum_{i=0}^{L-1} \sum_{j=0}^n C_{i,j}(x) \leq \sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j}$.
 - Dimostrare che il valore atteso di $\sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j}$ è $(2-1/2^n)L$.
 - Dimostrare che la probabilità che sia $\sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j} > 4L$ è al massimo $1/n^2$, usando il *limite di Chernoff* che afferma che, se X è la somma di un numero finito di variabili casuali booleane indipendenti, aventi valore atteso $\mu > 0$, allora $\text{Prob}(X > 2\mu) < (4/e)^{-\mu}$, dove $e = 2.71828128\dots$
 - Dedurre che l'algoritmo quick-sort probabilistico viene eseguito in un tempo $O(n \log n)$ con probabilità non inferiore a $1 - 1/n$.
- C-12.56 Possiamo rendere deterministico l'algoritmo quick-select, scegliendo il pivot di una sequenza di n elementi in questo modo:

Partiziona l'insieme S in $\lceil n/5 \rceil$ gruppi, ciascuno di dimensione 5 (tranne eventualmente un solo gruppo). Ordina ciascuno di questi piccoli insiemi e individua l'elemento mediano in ciascun insieme. In questo insieme di $\lceil n/5 \rceil$ "piccoli" valori mediani, applica ricorsivamente l'algoritmo di selezione per

trovare il valore mediano dei “piccoli” valori mediani. Usa questo elemento come pivot e procedi come nell’algoritmo quick-select standard.

Seguendo la traccia presentata nei cinque punti di questo esercizio, dimostrare che questo algoritmo quick-select deterministico viene eseguito in un tempo $O(n)$. Per semplificare la trattazione matematica, è possibile ignorare le funzioni “più piccolo intero non minore di” (*ceiling*) e “più grande intero non maggiore di” (*floor*), perché i risultati asintotici non cambiano.

- Quanti “piccoli” valori mediani sono non maggiori del pivot scelto? Quanti sono non minori del pivot?
- Per ogni “piccolo” valore mediano non maggior del pivot, quanti altri elementi sono non maggiori del pivot? Vale la stessa proprietà per quelli non minori del pivot?
- Dedurre che il metodo che trova il pivot in modo deterministico e lo usa per partizionare S viene eseguito in un tempo $O(n)$.
- Sulla base di queste stime, scrivere un’equazione di ricorrenza che ponga un limite superiore al tempo d’esecuzione $t(n)$ nel caso peggiore per questo algoritmo di selezione. Si osservi che, nel caso peggiore, vengono eseguite due ricorsioni: una per trovare il valore mediano tra i “piccoli” valori mediani e una per continuare ricorsivamente la ricerca in L o in G .
- Usando questa equazione di ricorrenza, dimostrare per induzione che $t(n)$ è $O(n)$.

C-12.57 Supponiamo di essere interessati alla gestione dinamica di un insieme S di numeri interi, inizialmente vuoto, con il supporto delle due operazioni seguenti:

add(v): Aggiunge il valore v all’insieme S .

median(): Restituisce il valore mediano dell’insieme. In un insieme avente cardinalità pari, definiamo il valore mediano come il valore medio tra i due elementi centrali.

Memorizzeremo ciascun elemento dell’insieme in una coda prioritaria scelta tra queste due: una coda prioritaria orientata all’elemento minimo, Q^+ , contenente tutti gli elementi non minori del valore mediano dell’insieme, e una coda prioritaria orientata all’elemento massimo, Q^- , contenente tutti gli elementi minori del valore mediano.

- Spiegare come si possa eseguire l’operazione **median()** in un tempo $O(1)$, data tale rappresentazione dell’insieme.
- Spiegare come si possa eseguire l’operazione **$S.add(k)$** in un tempo $O(\log n)$, dove n è la cardinalità dell’insieme, preservando le proprietà di tale rappresentazione.

C-12.58 Come generalizzazione del problema precedente, risolvere di nuovo l’Esercizio C-11.45, dove c’è bisogno di eseguire generiche operazioni di selezione in insiemi dinamici di valori.

Progettazione

- P-12.59 Implementare una versione non ricorsiva dell'algoritmo quick-sort che operi sul posto, così come è stata descritta alla fine del Paragrafo 12.2.2.
- P-12.60 Confrontare sperimentalmente le prestazioni di due versioni di quick-sort, una che operi sul posto e una che non operi sul posto.
- P-12.61 Eseguire una serie di verifiche di prestazioni per gli algoritmi merge-sort e quick-sort, al fine di determinare quale dei due sia più veloce. Tra le sequenze usate come prova devono essere presenti sia sequenze "casuali" sia sequenze "quasi ordinate".
- P-12.62 Implementare l'algoritmo quick-sort nella versione deterministica e in quella probabilistica, eseguendo poi una serie di verifiche di prestazioni per determinare quale delle due sia più veloce. Tra le sequenze usate come prova devono essere presenti sia sequenze "casuali" sia sequenze "quasi ordinate".
- P-12.63 Implementare una versione di insertion-sort e una versione di quick-sort che operino sul posto, eseguendo poi una serie di verifiche di prestazioni per determinare l'intervallo di valori di n per i quali l'algoritmo quick-sort è mediamente migliore dell'algoritmo insertion-sort.
- P-12.64 Progettare e implementare una versione dell'algoritmo bucket-sort che ordini una lista di n voci che abbiano come chiavi numeri interi appartenenti all'intervallo $[0, N - 1]$, con N intero e $N \geq 2$. L'algoritmo deve essere eseguito in un tempo $O(n + N)$.
- P-12.65 Implementare un'animazione di uno degli algoritmi di ordinamento descritti in questo capitolo, illustrando in modo intuitivo le sue proprietà.
- P-12.66 Progettare e implementare in Java due versioni dell'algoritmo bucket-sort, una per ordinare un array di valori di tipo `byte` e una per ordinare un array di valori di tipo `short`. Confrontare sperimentalmente le prestazioni delle sue implementazioni con quelle del metodo `java.util.Arrays.sort`.

Note

Il libro *Sorting and Searching* di Knuth [61], ormai un classico, contiene una ricerca storica sul problema dell'ordinamento e sugli algoritmi che lo risolvono. Huang e Langston [49] mostrano come fondere sul posto due liste ordinate in un tempo lineare. L'algoritmo quick-sort standard è dovuto a Hoare [45], e parecchie ottimizzazioni di tale algoritmo sono state descritte da Bentley e McIlroy [15]. Nel libro di Motwani e Raghavan [75] si possono trovare ulteriori informazioni relative agli algoritmi probabilistici. L'analisi dell'algoritmo quick-sort presentata in questo capitolo è una combinazione dell'analisi fornita in una precedente edizione di questo libro dedicato al linguaggio Java e dell'analisi di Kleinberg e Tardos [57]. L'Esercizio C-12.33 è stato proposto da Littman. Gonnet e Baeza-Yates [38] hanno analizzato e confrontato sperimentalmente molti algoritmi di ordinamento. Il termine "prune-and-search" viene dalla letteratura specifica della geometria computazionale (è presente, ad esempio, nei lavori di Clarkson [22] e Megiddo [70]). Il termine "decrease-and-conquer" è stato coniato da Levitin [66].

—

13

Elaborazione di testi

13.1 Abbondanza di testi digitalizzati

A dispetto della ricchezza di informazioni multimediali, l'elaborazione di testi rimane una delle funzioni principali dei calcolatori, che sono utilizzati per modificare, archiviare e visualizzare documenti, oltre che per trasferire i relativi file da una parte all'altra della rete Internet. Inoltre, molti sistemi digitali sono stati usati per archiviare una gran varietà di informazioni di tipo testo, e nuovi dati vengono generati a un ritmo rapidamente crescente. Una grande raccolta di questo tipo può facilmente superare un petabyte di dati, che è equivalente a mille terabyte o un milione di gigabyte. Tra i più diffusi esempi di archivi digitali che contengono informazioni in formato testuale possiamo citare:

- “Fotografie istantanee” (*snapshot*) del World Wide Web, sotto forma di file in formato HTML (il formato standard per i documenti presenti in Internet) e XML, con marcatori aggiuntivi per l'inserimento di contenuto multimediale
- Documenti archiviati localmente nel computer dei singoli utenti
- Archivi di messaggi di posta elettronica
- Raccolte di aggiornamenti di stato sui siti delle “reti sociali” (*social network*), come Facebook
- Flussi di aggiornamenti (*feed*) provenienti da siti di microblogging, come Twitter e Tumblr

Queste raccolte contengono testi scritti in centinaia di linguaggi internazionali. Oltre a questi, vanno citati i grandi insiemi di dati (come quelli relativi al DNA) che, dal punto di vista computazionale, possono essere considerati alla stregua di “stringhe di caratteri”, anche se non appartengono a un linguaggio.

In questo capitolo vedremo alcuni degli algoritmi fondamentali utilizzabili per analizzare ed elaborare in modo efficiente grandi insiemi di dati di tipo testuale. Oltre ad avere applicazioni interessanti, gli algoritmi di elaborazione di testi sono anche adatti a illustrare alcuni importanti schemi progettuali per algoritmi.

Iniziamo prendendo in esame il problema della ricerca di una stringa (detta *pattern*, cioè "schema", "esempio", "modello" o "campione") come sottostringa di una porzione di testo di dimensioni maggiori, come, ad esempio, la ricerca di una parola in un documento. Questo problema, chiamato del *pattern matching* (cioè della ricerca di una corrispondenza per un campione di stringa in un testo), può essere risolto con il cosiddetto *metodo della forza bruta (brute-force method)*, che risulta spesso inefficiente ma ha un campo di applicabilità generale. Continueremo descrivendo algoritmi più efficienti per risolvere il problema del *pattern matching*, esaminando anche alcune strutture dati aventi l'obiettivo specifico di organizzare in modo migliore i dati di tipo testo, allo scopo di consentire lo sviluppo di ricerche più efficienti.

A causa delle enormi dimensioni che spesso caratterizzano gli insiemi di dati di questo tipo, assume grande importanza il problema della compressione dei dati, tanto per minimizzare il numero di bit richiesto per trasferire i dati attraverso la rete quanto per ridurre l'occupazione di spazio a lungo termine all'interno degli archivi. Per la compressione dei testi, possiamo progettare algoritmi usando il *metodo greedy* ("goloso"), che spesso ci consente di trovare soluzioni approssimate a problemi difficili, e per alcuni problemi (come la compressione del testo) è anche in grado di generare algoritmi ottimali.

Infine, parleremo di *programmazione dinamica*, una tecnica algoritmica applicabile in determinate situazioni per risolvere un problema in un tempo polinomiale, mentre a prima vista sembrerebbe richiedere un tempo d'esecuzione esponenziale. Presenteremo un'applicazione di questa tecnica al problema di individuare corrispondenze parziali tra stringhe che possono essere simili ma non perfettamente allineate. Questo problema emerge quando, durante la scrittura di un documento, si cercano suggerimenti per una parola scritta in modo non corretto, oppure quando si cercano somiglianze tra campioni di materiale genetico.

13.1.1 Notazioni per stringhe di caratteri

Parlando di algoritmi per l'elaborazione di testi, si usano stringhe di caratteri per rappresentare, appunto, un modello per un testo. Le stringhe di caratteri possono provenire da una grande varietà di sorgenti di informazione, comprese diverse applicazioni scientifiche, linguistiche o per Internet. Ecco alcuni esempi di stringhe:

$$S = \text{"CGTAACTGCTTAATCAAACGC"}$$

$$T = \text{"http://www.wiley.com"}$$

La prima stringa, S , proviene da applicazioni di elaborazione del DNA, mentre la seconda, T , è l'indirizzo Internet (URL, *uniform resource locator*) dell'editore della versione originale di questo libro.

Per consentire nelle descrizioni dei nostri algoritmi l'utilizzo di un concetto di stringa sufficientemente generale, ipotizziamo soltanto che i caratteri appartenenti a una stringa provengano da un alfabeto noto, che indichiamo con Σ . Ad esempio, nel contesto di applicazioni che riguardano il DNA, l'alfabeto standard è costituito da quattro simboli, $\Sigma =$

$\{A, C, G, T\}$. L'alfabeto Σ può, ovviamente, essere un sottoinsieme dell'insieme di caratteri ASCII o Unicode, ma potrebbe anche essere più generale. Anche se ipotizziamo che un alfabeto abbia una dimensione finita e prefissata, indicata con $|\Sigma|$, tale dimensione può essere rilevante, come nel caso dell'utilizzo, nel linguaggio di programmazione Java, dell'alfabeto Unicode, che contiene più di un milione di caratteri diversi. Nell'analisi asintotica degli algoritmi di elaborazione di testi, quindi, prenderemo in considerazione l'influenza di $|\Sigma|$.

Come detto nel Paragrafo 1.3, la classe `String` di Java fornisce supporto alla rappresentazione di una sequenza *immutable* di caratteri, mentre la classe `StringBuilder` viene usata per rappresentare sequenze di caratteri *modificabili*. Nella maggior parte di questo capitolo, ci baseremo su una rappresentazione più primitiva delle stringhe come array di `char`, principalmente perché questo consente l'utilizzo della notazione standard mediante indice, come $S[i]$, invece della sintassi decisamente più ingombrante che sarebbe richiesta dalla classe `String`, con espressioni come `S.charAt(1)`.

Per indicare porzioni di stringa, introduciamo il concetto di *sottostringa (substring)* di una stringa P avente n caratteri come di una stringa descritta nella forma $P[i]P[i+1]P[i+2]\dots P[j]$, con $0 \leq i \leq j \leq n - 1$. Per semplificare la notazione, indicheremo con $P[i..j]$ la sottostringa di P che va dall'indice i all'indice j , entrambi inclusi. Osserviamo che una stringa è, dal punto di vista tecnico, una sottostringa di se stessa (prendendo $i = 0$ e $j = n - 1$), quindi, quando vogliamo escludere questa possibilità, dobbiamo restringere la definizione alle sottostringhe *proprie*, nelle quali $i > 0$ oppure $j < n - 1$. Per convenzione, poi, ammettiamo che, quando $i > j$, la sottostringa $P[i..j]$ sia uguale alla *stringa nulla*, che ha lunghezza 0.

Inoltre, per introdurre alcune categorie di stringhe speciali, quando una sottostringa ha la forma $P[0..j]$, con $0 \leq j \leq n - 1$, la chiamiamo *prefisso* di P , mentre una sottostringa come $P[i..n - 1]$, con $0 \leq i \leq n - 1$, è un *suffisso* di P . Ad esempio, se consideriamo di nuovo che P sia la stringa di DNA presentata all'inizio del paragrafo, allora "CGTAA" è un prefisso di P , "CGC" è un suffisso di P e "TTAAC" è una sottostringa (propria) di P . Osserviamo che la stringa nulla è sia un prefisso sia un suffisso di qualsiasi altra stringa.

13.2 Algoritmi di pattern matching

Nel problema del pattern matching classico, ci viene fornita una stringa di *testo* di lunghezza n e una stringa, che svolge il ruolo di *pattern* da cercare, di lunghezza $m \leq n$: dobbiamo determinare se il *pattern* sia una sottostringa del *testo*. Se lo è, possiamo voler trovare il minimo indice all'interno del testo che rappresenta l'inizio di una occorrenza del pattern, oppure *tutti* gli indici, nel testo, in cui inizia un'occorrenza del pattern.

Il problema del pattern matching è inherente a molti dei comportamenti della classe `String`, come `text.contains(pattern)` e `text.indexOf(pattern)`, ed è anche un sotto-problema di operazioni più complesse di elaborazione di stringhe, come `text.replace(pattern, substitute)` e `text.split(pattern)`.

In questo paragrafo presenteremo tre algoritmi di pattern matching, con livello di difficoltà crescente. Le nostre implementazioni restituiscono l'indice in cui inizia l'occorrenza del pattern più a sinistra nel testo, se ce n'è almeno una. In caso di ricerca infruttuosa, adottiamo la convenzione prevista dal metodo `indexOf` della classe `String` di Java, restituendo -1 come valore sentinella.

13.2.1 Forza bruta

Lo schema che consente di progettare algoritmi mediante *forza bruta* è una potente tecnica di progettazione di algoritmi utilizzabile quando vogliamo cercare qualcosa o quando vogliamo ottimizzare una determinata funzione. In generale, applicando questa tecnica tipicamente elenchiamo tutte le possibili configurazioni dei dati da elaborare e, tra queste, scegliamo la migliore di tutte.

Applicando questa tecnica alla progettazione di un algoritmo di pattern matching, otteniamo quello che probabilmente è il primo algoritmo che ci viene in mente per risolvere il problema: semplicemente, verifichiamo tutti i possibili posizionamenti del pattern all'interno del testo. Il Codice 13.1 mostra un'implementazione di questo algoritmo.

Codice 13.1: Un'implementazione dell'algoritmo di pattern matching mediante forza bruta (per semplificare la notazione, usiamo array di caratteri invece di stringhe).

```

1  /** Restituisce l'indice minimo in cui inizia pattern nel testo (oppure -1). */
2  public static int findBrute(char[] text, char[] pattern) {
3      int n = text.length;
4      int m = pattern.length;
5      for (int i=0; i <= n - m; i++) { // prova tutti gli indici iniziali nel testo
6          int k = 0; // k è un indice nel pattern
7          while (k < m && text[i+k] == pattern[k]) // il k-esimo carattere corrisponde
8              k++;
9          if (k == m) // se abbiamo raggiunto la fine del pattern
10             return i; // la sottostringa text[i..i+m-1] corrisponde al pattern
11     }
12     return -1; // ricerca fallita
13 }
```

Prestazioni

L'analisi dell'algoritmo di pattern matching mediante forza bruta non potrebbe essere più semplice. È costituito da due cicli annidati, con il ciclo esterno che scandisce tutti i possibili indici iniziali del pattern nel testo e il ciclo interno che analizza ciascun carattere del pattern, confrontandolo con il carattere potenzialmente corrispondente nel testo. Quindi, la correttezza di questo algoritmo discende immediatamente da questo approccio di ricerca esaustivo.

Tuttavia, nel caso peggiore, il tempo d'esecuzione dell'algoritmo di pattern matching mediante forza bruta non è buono, perché per ogni verifica di un possibile allineamento del pattern nel testo posso essere necessari fino a m confronti tra coppie di caratteri. Faccendo riferimento al Codice 13.1, vediamo che il ciclo `for`, quello esterno, viene eseguito al massimo $n - m + 1$ volte, mentre il ciclo `while`, quello interno, viene eseguito al massimo m volte. Quindi, il tempo d'esecuzione di questo algoritmo, nel caso peggiore, è $O(nm)$.

Esempio 13.1: Supponiamo che, in un problema di pattern matching, il testo sia la stringa seguente:

```
text = "abacaabaccabacabaabb"
```

e il pattern sia:

```
pattern = "abacab"
```

La Figura 13.1 mostra l'esecuzione dell'algoritmo di pattern matching mediante forza bruta per questo problema.

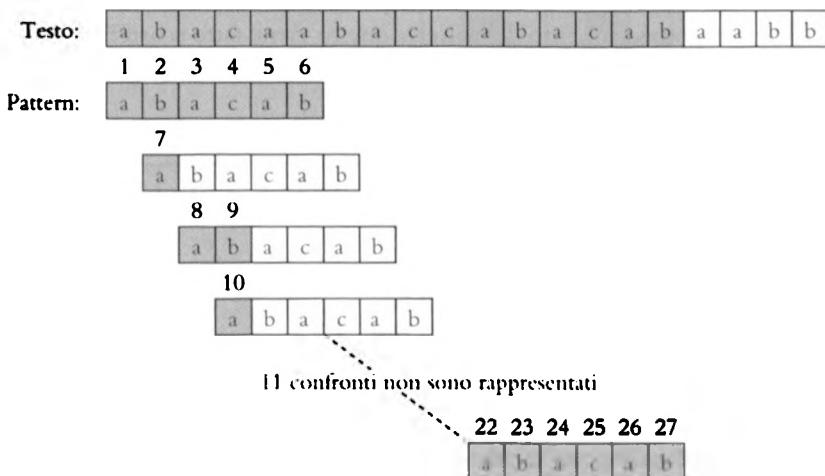


Figura 13.1: Esempio di esecuzione dell'algoritmo di pattern matching mediante forza bruta. L'algoritmo effettua 27 confronti tra caratteri, indicati con etichette numerate.

13.2.2 L'algoritmo di Boyer-Moore

A prima vista si potrebbe pensare che, per poter individuare il pattern come sottostringa di un testo o per decidere sulla sua assenza, sia sempre necessario esaminare tutti i caratteri del testo, ma non è così. L'algoritmo di *Boyer-Moore* per il pattern matching, che studieremo in questo paragrafo in una versione semplificata, a volte può evitare l'esame di una frazione rilevante dei caratteri del testo.

L'idea principale su cui si basa l'algoritmo di Boyer-Moore è quella di migliorare il tempo d'esecuzione dell'algoritmo a forza bruta aggiungendo due strategie euristiche che possono far risparmiare tempo. In sintesi, si tratta di questo:

Euristica Looking-Glass: (euristica che “guarda allo specchio”) Quando si verifica un possibile posizionamento del pattern nel testo, i confronti tra le coppie di caratteri vengono effettuati procedendo da destra a sinistra.

Euristica Character-Jump: (euristica che “salta dei caratteri”) Durante la verifica di un possibile posizionamento del pattern nel testo, una differenza tra il carattere $\text{text}[i] = c$ e il corrispondente carattere $\text{pattern}[k]$ viene gestita in questo modo: se c non è presente in alcuna posizione del pattern, si fa scorrere il pattern completamente a destra della posizione $\text{text}[i]$; altrimenti, si fa scorrere il pattern verso destra finché l'occorrenza di c che si trova più a destra nel pattern risulta essere allineata con $\text{text}[i]$.

A breve definiremo in modo più formale queste strategie euristiche, ma, a livello intuitivo, collaborano tra loro, come in una squadra, per consentirci di evitare confronti con intere sequenze di caratteri all'interno del testo. In particolare, quando si trova una mancata

corrispondenza tra caratteri in una posizione vicina alla fine del pattern, possiamo trovarci a riallineare il pattern oltre tale posizione, senza dover mai esaminare i molti caratteri del testo che precedono la posizione in questione. Ad esempio, la Figura 13.2 mostra alcune applicazioni di queste euristiche. Si osservi che, quando i caratteri *e* e *i* non corrispondono, alla fine del primo posizionamento del pattern, questo viene fatto scorrere completamente oltre la posizione di mancata corrispondenza, evitando così l'esame dei primi quattro caratteri del testo.

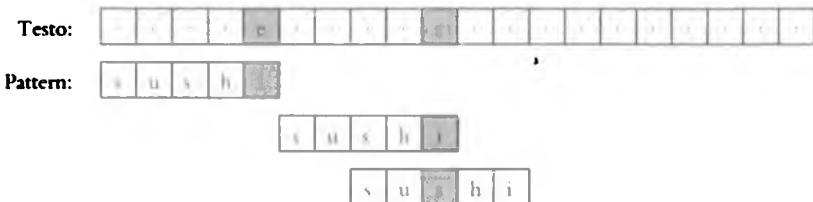


Figura 13.2: Un semplice esempio che illustra l'intuizione su cui si basa l'algoritmo di pattern matching di Boyer-Moore. Il primo confronto evidenzia una mancata corrispondenza con il carattere *e* del testo. Dato che quel carattere non è presente in nessuna posizione del pattern, l'intero pattern viene fatto scorrere oltre quella posizione. Il secondo confronto è ancora una mancata corrispondenza, ma il carattere del testo che non ha trovato corrispondenza è *s*, che compare nel pattern. Quindi, il pattern viene fatto scorrere in modo che l'ultima occorrenza di *s* nel pattern si trovi allineata con la *s* che si stava esaminando nel testo. La parte restante della procedura non è rappresentata nella figura.

L'esempio della Figura 13.2 è abbastanza semplice, perché riguarda solamente corrispondenze mancate con l'ultimo carattere del pattern. Più in generale, quando l'ultimo carattere del pattern trova invece corrispondenza nel testo, l'algoritmo procede cercando di estendere la porzione identica, passando al penultimo carattere del pattern senza cambiare il suo posizionamento rispetto al testo. Il processo continua finché si trova nel testo una corrispondenza per l'intero pattern oppure si trova una coppia di caratteri diversi in qualche posizione del pattern diversa dall'ultima.

Se si trova una mancata corrispondenza tra una coppia di caratteri e il carattere preso in esame nel testo non ricorre in alcuna posizione del pattern, facciamo scorrere l'intero pattern a destra, oltre tale posizione esaminata, come avviene nel primo caso della Figura 13.2. Se, invece, il carattere preso in esame nel testo è presente in qualche posizione del pattern, dobbiamo considerare due possibili sotto-casi, in relazione al fatto che la sua occorrenza più a destra (cioè, come si dice, l'*ultima* occorrenza) si trovi a sinistra o a destra del carattere che, nel pattern, è stato esaminato per ultimo e ha provocato la mancata corrispondenza. Questi due casi sono proprio illustrati nella Figura 13.3.

Nel caso della Figura 13.3(b), facciamo scorrere il pattern di una sola posizione. Sarebbe più efficace farlo scorrere verso destra fino a quando non si trova nel pattern un'altra occorrenza del carattere $\text{text}[i]$, ma non vogliamo cercare tale altra occorrenza. L'efficienza dell'algoritmo di Boyer-Moore sta nel fatto che si possa determinare velocemente se un carattere che ha provocato una mancata corrispondenza è presente da qualche parte nel pattern. In particolare, definiamo la funzione $\text{last}(c)$ in questo modo:

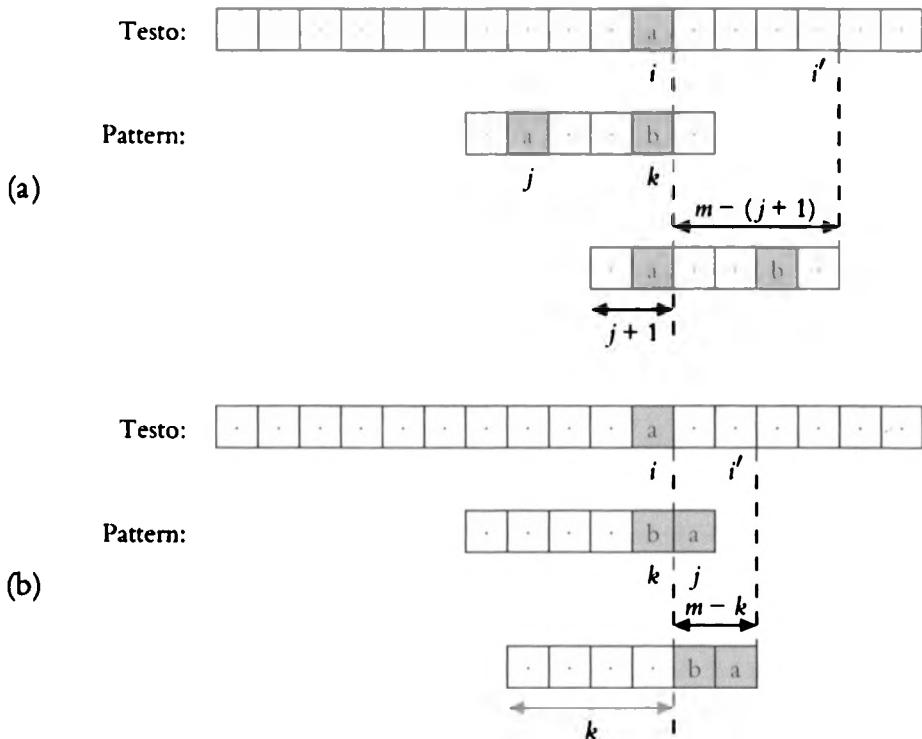


Figura 13.3: Ulteriori regole per l'euristica *character-jump* dell'algoritmo Boyer-Moore.

Indichiamo con i l'indice, nel testo, del carattere che ha provocato la mancata corrispondenza, mentre k è l'indice corrispondente nel pattern e j rappresenta l'indice della posizione più a destra di $\text{text}[i]$ nel pattern. Distinguiamo due casi: (a) $j < k$, nel qual caso facciamo scorrere il pattern di $k - j$ posizioni e, quindi, l'indice i aumenta di $m - (j + 1)$ unità; (b) $j > k$, nel qual caso facciamo scorrere il pattern di una sola posizione e, quindi, l'indice i aumenta di $m - k$ unità.

- Se c appartiene al pattern, $\text{last}(c)$ è l'indice dell'occorrenza più a destra di c nel pattern (cioè l'*ultima*, appunto "last"); altrimenti, definiamo per convenzione $\text{last}(c) = -1$.

Ipotizzando che l'alfabeto abbia dimensione finita e prefissata e che i caratteri possano essere usati come indici in un array (ad esempio, usando il codice corrispondente al carattere), si può facilmente implementare la funzione last come una tabella di ricerca, con un tempo d'accesso al valore $\text{last}(c)$ che, nel caso peggiore, è $O(1)$. La tabella, però, avrebbe una dimensione uguale alla dimensione dell'alfabeto (non correlata alla dimensione del pattern) e ci vorrebbe tempo per inizializzare l'intera tabella.

Preferiamo, quindi, rappresentare la funzione last con una tabella hash, che realizza una mappa contenente soltanto i caratteri effettivamente presenti nel pattern. Lo spazio di memoria utilizzato da questo approccio è proporzionale al numero di simboli dell'alfabeto distinti che compaiono nel pattern, quindi è $O(\max(m, |\Sigma|))$. Il tempo atteso per la ricerca rimane $O(1)$, così come il suo caso peggiore, se consideriamo che $|\Sigma|$ sia un valore costante. Il Codice 13.2 presenta la nostra implementazione completa dell'algoritmo di pattern matching di Boyer-Moore.

Codice 13.2: Un'implementazione dell'algoritmo di Boyer-Moore.

```

1  /** Restituisce l'indice minimo in cui inizia pattern nel testo (oppure -1). */
2  public static int findBoyerMoore(char[] text, char[] pattern) {
3      int n = text.length;
4      int m = pattern.length;
5      if (m == 0) return 0;          // ricerca banale di una stringa vuota
6      Map<Character, Integer> last = new HashMap<>(); // la mappa della funzione 'last'
7      for (int i=0; i < n; i++)
8          last.put(text[i], -1);    // usa -1 come default per tutti i caratteri del testo
9      for (int k=0; k < m; k++)
10         last.put(pattern[k], k); // l'occorrenza più a destra sarà l'ultima memorizzata
11     // inizia con la fine del pattern allineata con l'indice m-1 del testo
12     int i = m-1;                // indice nel testo
13     int k = m-1;                // indice nel pattern
14     while (i < n) {
15         if (text[i] == pattern[k]) { // trovato un carattere che corrisponde
16             if (k == 0) return i;    // trovato un intero pattern che corrisponde
17             i--;                  // altrimenti, esamina il carattere precedente
18             k--;                  // tanto nel testo quanto nel pattern
19         } else {
20             i += m - Math.min(k, i + last.get(text[i])); // decide come fare il salto
21             k = m - 1;            // riparte dalla fine del pattern
22         }
23     }
24     return -1;                  // ricerca fallita
25 }
```

La correttezza dell'algoritmo di pattern matching di Boyer-Moore discende dal fatto che, ogni volta che il metodo fa scorrere il pattern in avanti, si ha la garanzia di non "saltare" la posizione di una possibile corrispondenza completa tra il pattern e il testo, cioè, come anche si dice, un *match*, perché la funzione *last(c)* indica la posizione dell'*ultima* occorrenza di *c* nel pattern. Nella Figura 13.4 illustriamo l'esecuzione dell'algoritmo di pattern matching di Boyer-Moore applicato a una stringa simile a quella dell'Esempio 13.1.

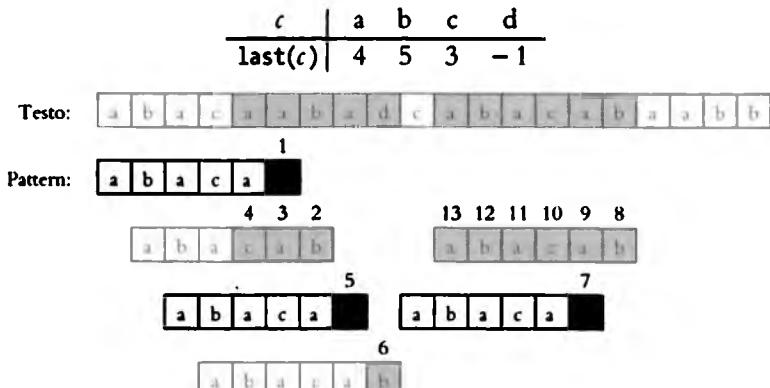


Figura 13.4: Rappresentazione grafica dell'esecuzione dell'algoritmo di pattern matching di Boyer-Moore; è riportata anche la funzione *last(c)*. L'algoritmo effettua 13 confronti tra coppie di caratteri, indicati con etichette numerate.

Prestazioni

Se si usa una tabella di ricerca tradizionale, il caso peggiore del tempo d'esecuzione dell'algoritmo di Boyer-Moore è $O(nm + |\Sigma|)$. Il calcolo della funzione *last* richiede un tempo $O(m + |\Sigma|)$, sebbene la dipendenza da $|\Sigma|$ possa essere rimossa usando una tabella hash. La ricerca di un pattern richiede un tempo $O(nm)$ nel caso peggiore, esattamente come avviene con l'algoritmo a forza bruta. Un esempio che costituisce un caso peggiore per Boyer-Moore è questo:

$$\begin{aligned} \text{testo} &= \overbrace{aaaaaa \cdots a}^n \\ \text{pattern} &= b \overbrace{aa \cdots a}^{m-1} \end{aligned}$$

La situazione del caso peggiore, però, accade molto raramente se si fanno ricerche in un testo in lingua inglese: in quel caso, l'algoritmo di Boyer-Moore è spesso in grado di saltare porzioni di testo di grandi dimensioni. Esperimenti condotti su testi in lingua inglese evidenziano come il numero medio di confronti effettuati per ciascun carattere del testo sia 0.24 nel caso di un pattern di lunghezza cinque.

Abbiamo presentato una versione semplificata dell'algoritmo di Boyer-Moore. L'algoritmo originale è in grado di ottenere un tempo d'esecuzione che, nel caso peggiore, è $O(n + m + |\Sigma|)$, usando una strategia di scorrimento euristica alternativa per le porzioni di testo che costituiscono una corrispondenza parziale con il pattern, con efficacia maggiore dell'euristica *character-jump* illustrata qui. Questa strategia alternativa è basata sull'applicazione della stessa idea fondamentale che caratterizza l'algoritmo di pattern matching dovuto a Knuth, Morris e Pratt, di cui parleremo ora.

13.2.3 L'algoritmo di Knuth-Morris-Pratt

Analizzando le prestazioni nel caso peggiore degli algoritmi di pattern matching di Boyer-Moore e a forza bruta applicati a esemplari specifici del problema, come quello dell'Esempio 13.1, dovrebbe risultare evidente una grande inefficienza (almeno nel caso peggiore): per alcuni posizionamenti del pattern, anche se riscontriamo parecchi caratteri corretti prima di trovare un carattere non corrispondente, ignoriamo poi tutte le informazioni ottenute dai confronti andati a buon fine e ripartiamo semplicemente con il posizionamento successivo del pattern.

L'algoritmo KMP (dalle iniziali degli inventori, Knuth-Morris-Pratt), discusso in questo paragrafo, evita questo spreco di informazioni e, così facendo, riesce a raggiungere prestazioni $O(n + m)$ nel caso peggiore, che sono assintoticamente ottime, perché, nel caso peggiore, qualunque algoritmo di pattern matching dovrà esaminare almeno una volta tutti i caratteri del testo e tutti i caratteri del pattern. L'idea principale su cui si basa l'algoritmo KMP è quella di pre-calcolare le auto-sovrapposizioni tra porzioni del pattern in modo che, quando in una determinata posizione si riscontra una mancata corrispondenza, si possa sapere immediatamente lo spostamento massimo verso destra a cui si può sottoporre il pattern prima di continuare la ricerca. La Figura 13.5 mostra un esempio che motiva questo approccio.

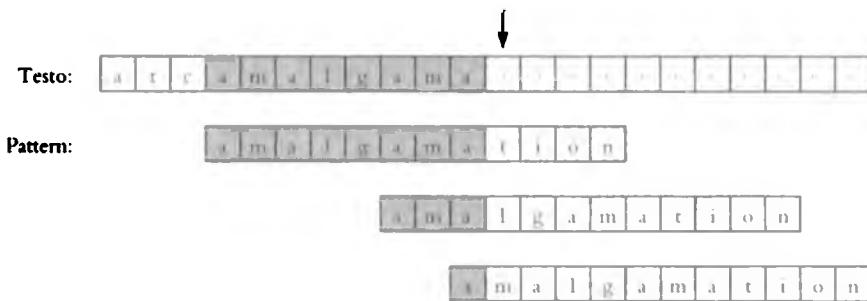


Figura 13.5: Un esempio che motiva la strategia utilizzata dall'algoritmo KMP. Se nella posizione indicata dalla freccia si riscontra una mancata corrispondenza tra il carattere del pattern e quello del testo, si può far scorrere il pattern fino a raggiungere la seconda configurazione, senza che ci sia bisogno di verificare di nuovo in modo esplicito la corrispondenza parziale che era stata individuata nel prefisso *ama*. Se il carattere del testo posto in corrispondenza della freccia (che aveva causato il primo errore di matching) non è un carattere 1, allora il prossimo posizionamento del pattern, alla ricerca di un potenziale allineamento, può trarre vantaggio dal carattere a condiviso tra la parte iniziale del pattern e il testo.

La funzione fallimento

Per implementare l'algoritmo KMP, dovremo pre-calcolare la cosiddetta *funzione fallimento* (*failure function*), f , che fornisce il valore corretto per lo scorrimento verso destra del pattern in caso di fallimento di un confronto. Nello specifico, la funzione fallimento $f(k)$ è definita come la dimensione del più lungo prefisso del pattern che è anche suffisso della sottostringa $\text{pattern}[1..k]$ (si osservi che *non* abbiamo incluso il carattere $\text{pattern}[0]$ nella sottostringa, perché faremo certamente scorrere il pattern almeno di una posizione). Intuitivamente, se riscontriamo un confronto fallito per il carattere $\text{pattern}[k+1]$, la funzione $f(k)$ ci dice quanti dei caratteri consecutivi immediatamente precedenti a $\text{pattern}[k+1]$ possono essere riutilizzati come "validi" per far ripartire il confronto da una posizione più avanzata della prima. L'Esempio 13.2 descrive il valore della funzione fallimento per il pattern usato nell'esempio della Figura 13.5.

Esempio 13.2: Consideriamo il pattern $P = \text{"amalgamation"}$ della Figura 13.5. La funzione fallimento dell'algoritmo KMP, $f(k)$, per la stringa P è quella presentata nella tabella seguente:

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	.	l	g	a	m	a	t	i	o
$f(k)$	0	0	1	0	0	0	1	2	3	0	0	0

Implementazione

Il Codice 13.3 presenta la nostra implementazione dell'algoritmo di pattern matching KMP: si basa su un metodo ausiliario, `computeFailKMP`, che calcola in modo efficiente la funzione fallimento, come vedremo in seguito.

La parte principale dell'algoritmo KMP è il suo ciclo `while`, ciascuna iterazione del quale esegue un confronto tra il carattere che si trova nel testo all'indice j e il carattere del pattern all'indice k . Se il risultato di questo confronto è una corrispondenza, l'algoritmo si sposta

al carattere successivo tanto nel testo quanto nel pattern (o restituisce una segnalazione di *match* se ha raggiunto la fine del pattern). Se, invece, il confronto dà esito negativo, l'algoritmo consulta la funzione fallimento per trovare nel pattern un nuovo carattere candidato al confronto, oppure, se il fallimento è avvenuto in corrispondenza del primo carattere del pattern, riparte dall'indice successivo nel testo, perché non ci sono informazioni da riutilizzare.

Codice 13.3: Un'implementazione dell'algoritmo di pattern matching KMP. Il metodo ausiliario `computeFailKMP` è descritto nel Codice 13.4.

```

1  /** Restituisce l'indice minimo in cui inizia pattern nel testo (oppure -1). */
2  public static int findKMP(char[] text, char[] pattern) {
3      int n = text.length;
4      int m = pattern.length;
5      if (m == 0) return 0;          // ricerca banale di una stringa vuota
6      int[] fail = computeFailKMP(pattern); // funzione calcolata da un metodo privato
7      int j = 0;                  // indice nel testo
8      int k = 0;                  // indice nel pattern
9      while (j < n) {
10          if (text[j] == pattern[k]) { // il pattern[0..k] corrisponde fin qui
11              if (k == m - 1) return j - m + 1; // corrispondenza completa
12              j++;                      // altrimenti cerca di estendere la corrispondenza
13              k++;
14          } else if (k > 0)
15              k = fail[k - 1];        // riutilizza il suffisso di P[0..k-1]
16          else
17              j++;
18      }
19      return -1;                  // ricerca fallita
20  }
```

Costruzione della funzione fallimento di KMP

Per costruire la funzione fallimento usiamo il metodo presentato nel Codice 13.4, che è una sorta di processo “di bootstrap” (che “sta in piedi da solo”...) che confronta il pattern con se stesso secondo quanto previsto dall’algoritmo KMP. Ogni volta che due caratteri corrispondono, eseguiamo l’assegnazione $f(j) = k + 1$. Osserviamo che, essendo $j > k$ durante l’intera esecuzione dell’algoritmo, il valore di $f(k - 1)$ è sempre già stato definito nel momento in cui viene utilizzato.

Codice 13.4: Un’implementazione del metodo ausiliario `computeFailKMP`, a supporto dell’algoritmo di pattern matching KMP. Si osservi che l’algoritmo utilizza i valori della funzione fallimento calcolati in precedenza per calcolare in modo efficiente i nuovi valori.

```

1  private static int[] computeFailKMP(char[] pattern) {
2      int m = pattern.length;
3      int[] fail = new int[m]; // tutte le sovrapposizioni sono così uguali a zero
4      int j = 1;
5      int k = 0;
6      while (j < m) {           // in questo passo calcola fail[j], se non è zero
7          if (pattern[j] == pattern[k]) { // finora k+1 caratteri corrispondono
8              fail[j] = k + 1;
9              j++;
10             k++;
11         } else if (k > 0)       // k segue un prefisso che corrisponde
```

```

    k = fail[k - 1];
else
    j++;
}
return fail;
}
}

```

Prestazioni

Escludendo il calcolo della funzione fallimento, il tempo d'esecuzione dell'algoritmo KMP è chiaramente proporzionale al numero di iterazioni eseguite dal ciclo `while`. Per agevolare l'analisi, definiamo $s = j - k$: intuitivamente, s è lo spostamento complessivo del pattern verso destra rispetto alla posizione iniziale. Osserviamo che, durante l'intera esecuzione dell'algoritmo, si ha $s \leq n$. Durante ciascuna iterazione del ciclo si verifica uno dei tre casi seguenti:

- Se $\text{text}[j] = \text{pattern}[k]$, allora j e k aumentano di un'unità, lasciando così inalterato s .
- Se $\text{text}[j] \neq \text{pattern}[k]$ e $k > 0$, allora j non cambia e s aumenta almeno di un'unità, perché in questo caso s passa da $j - k$ a $j - f(k - 1)$; osserviamo che questo significa un aumento di s di una quantità uguale a $k - f(k - 1)$, che è certamente positiva perché $f(k - 1) < k$.
- Se $\text{text}[j] \neq \text{pattern}[k]$ e $k = 0$, allora j aumenta di un'unità e s diminuisce di un'unità, perché k non cambia.

Quindi, durante ciascuna iterazione del ciclo, j o s aumenta di almeno un'unità (eventualmente aumentano entrambi); di conseguenza, il numero totale di iterazioni del ciclo `while` dell'algoritmo di pattern matching KMP è al massimo $2n$. Per ottenere questo risultato, però, bisogna ovviamente aver già calcolato la funzione di fallimento relativa al pattern.

L'algoritmo che calcola la funzione di fallimento richiede un tempo $O(m)$ e la sua analisi è analoga a quella dell'algoritmo KMP principale, anche se con un pattern di lunghezza m confrontato con se stesso. Abbiamo, quindi:

Proposizione 13.3: *L'algoritmo KMP risolve il problema del pattern matching, cercando un pattern di lunghezza m in un testo di lunghezza n , in un tempo $O(n + m)$.*

La correttezza di questo algoritmo discende dalla definizione stessa di funzione fallimento. Qualunque confronto che venga evitato è effettivamente non necessario, perché la funzione fallimento garantisce che tutti i confronti ignorati siano ridondanti: si tratterebbe di rifare confronti già fatti, con esito positivo, tra gli stessi caratteri.

Nella Figura 13.6 mostriamo l'esecuzione dell'algoritmo di pattern matching KMP applicato agli stessi dati (testo e pattern) dell'Esempio 13.1. Si noti l'uso della funzione fallimento per evitare di rifare alcuni confronti tra un carattere del pattern e un carattere del testo. Si noti, ancora, che l'algoritmo esegue complessivamente un numero di confronti inferiore a quello richiesto dall'algoritmo a forza bruta eseguito con le stesse stringhe (si veda la Figura 13.1).

k	0	1	2	3	4	5
Funzione di fallimento: pattern [k]	a	b	a	c	a	b
$f(k)$	0	0	1	0	1	2

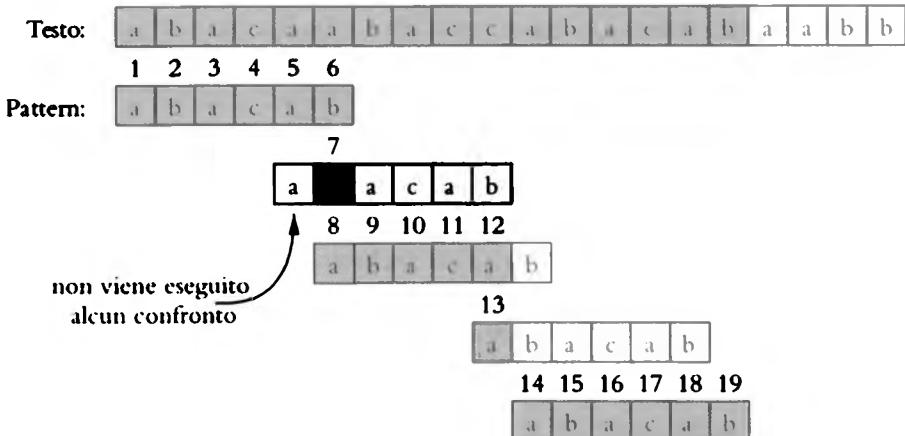


Figura 13.6: Visualizzazione del funzionamento dell'algoritmo di pattern matching KMP. L'algoritmo principale esegue 19 confronti tra caratteri, indicati con etichette numeriche (durante il calcolo della funzione fallimento vengono eseguiti altri confronti).

13.3 Trie

Gli algoritmi di pattern matching presentati nel Paragrafo 13.2 rendono più veloce la ricerca in un testo effettuando un'elaborazione preventiva del pattern (nell'algoritmo Boyer-Moore per calcolare la funzione *last* e nell'algoritmo KMP per calcolare la funzione fallimento). In questo paragrafo vedremo un approccio complementare: presentiamo algoritmi di pattern matching che elaborano preventivamente il testo, piuttosto che il pattern. Questo approccio è adatto a quelle applicazioni che eseguono ripetute ricerche all'interno di un testo prefissato, così che il costo iniziale della pre-elaborazione del testo sia compensato da un aumento della velocità di ciascuna ricerca successiva: ad esempio, un sito web che offre la possibilità di cercare pattern all'interno dell'opera *Hamlet* di Shakespeare, oppure un motore di ricerca che presenti all'utente pagine web contenenti il termine *Hamlet*.

Un *trie* (parola pronunciata come l'inglese "try" ma individuata come porzione della parola "retrieval") è una struttura dati ad albero per memorizzare stringhe al fine di agevolare un pattern matching veloce. L'applicazione principale per i tries è nel settore dell'*information retrieval* (da cui il nome), che significa "recupero dell'informazione" da una base di dati: un esempio tipico è la ricerca di una determinata sequenza di DNA all'interno di una base di dati genomica, che è costituita da una raccolta S di stringhe, tutte definite usando lo stesso alfabeto. Il tipo di interrogazione principale messo a disposizione da un trie è il pattern matching e il *prefix matching*: quest'ultima operazione prevede di cercare, tra tutte le stringhe appartenenti a un insieme S , quelle che iniziano con la stringa X , cioè che hanno X come prefisso.

13.3.1 Trie standard

Sia S un insieme di s stringhe, definite sull'alfabeto Σ , con la proprietà aggiuntiva che nessuna stringa di S sia un prefisso di un'altra stringa di S . Un *trie standard* per S è un albero ordinato T con le seguenti proprietà (illustrate nella Figura 13.7):

- Ogni nodo di T , tranne la radice, è etichettato con un carattere di Σ .
- I figli di un nodo interno di T hanno etichette distinte.
- T ha s foglie, ciascuna associata a una stringa di S , in modo che la concatenazione delle etichette dei nodi che si trovano lungo il percorso che va dalla radice a una foglia v di T generi la stringa di S che è associata a v .

Quindi, un trie T rappresenta le stringhe di S mediante percorsi che vanno dalla radice a una foglia di T . Si noti l'importanza di aver ipotizzato che nessuna stringa di S sia un prefisso di un'altra stringa di S : questo garantisce che ciascuna stringa di S sia associata in modo univoco a una foglia di T (questa ipotesi è simile al vincolo sui codici dei prefissi nella codifica di Huffman, che sarà descritta nel Paragrafo 13.4). Possiamo sempre rendere vera questa ipotesi aggiungendo alla fine di ciascuna stringa un carattere speciale che non appartenga all'alfabeto originario Σ .

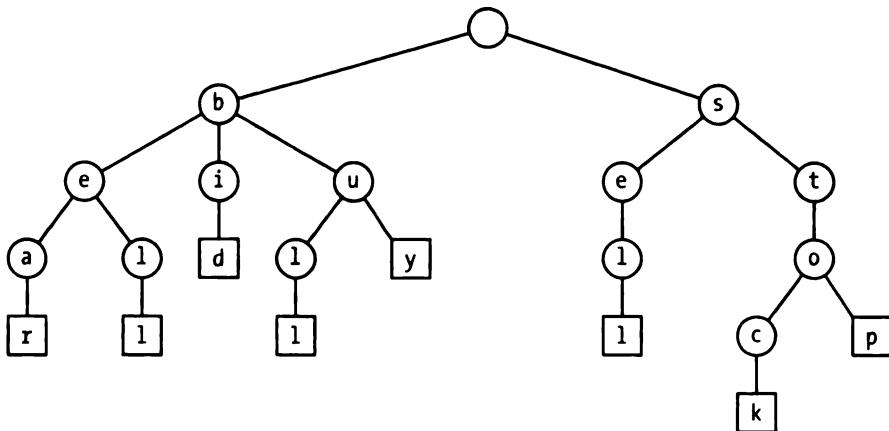


Figura 13.7: Trie standard per l'insieme di stringhe {bear, bell, bid, bull, buy, sell, stop}.

Un nodo interno di un trie standard T può avere un numero di figli qualsiasi, compreso tra 1 e $|\Sigma|$. Per ogni carattere che è il carattere iniziale in almeno una stringa di S esiste un ramo che va dalla radice r a uno dei suoi figli. Inoltre, un percorso che vada dalla radice di T a un nodo interno v di profondità k corrisponde a un prefisso di lunghezza k , $X[0..k - 1]$, di una stringa X di S . Inoltre, per ogni carattere c che può seguire il prefisso $X[0..k - 1]$ in una delle stringhe dell'insieme S , esiste un figlio di v etichettato con c . In questo modo, un trie memorizza in modo sintetico i prefissi comuni esistenti all'interno di un insieme di stringhe.

Come caso speciale, se nell'alfabeto esistono soltanto due caratteri, il trie è fondamentalmente un albero binario, con alcuni nodi interni che possono avere un solo figlio

(cioè può essere un albero binario improprio). In generale, pur essendo possibile che un nodo interno abbia fino a $|\Sigma|$ figli, in pratica il grado medio di tali nodi è probabilmente molto inferiore: ad esempio, il trie della Figura 13.7 ha molti nodi interni con un solo figlio. Per insiemi di dati di maggiori dimensioni, è probabile che il grado medio dei nodi diventi minore nei livelli più profondi dell'albero, perché saranno sempre meno le stringhe che condividono un prefisso sempre più lungo e, quindi, sempre minori le possibilità di allungare quel prefisso. Inoltre, in molti linguaggi, in particolare in quelli naturali, esistono combinazioni di caratteri il cui verificarsi è veramente improbabile.

La proposizione seguente elenca alcune proprietà strutturali importanti di un trie standard.

Proposizione 13.4: *Un trie standard T che memorizza una raccolta S di s stringhe di lunghezza totale n i cui caratteri appartengono all'alfabeto Σ gode delle seguenti proprietà:*

- L'altezza di T è uguale alla lunghezza della più lunga stringa presente in S .
- Ogni nodo interno di T ha al massimo $|\Sigma|$ figli.
- T ha s foglie.
- Il numero di nodi di T è al massimo $n + 1$.

Il caso peggiore per il numero di nodi di un trie si verifica quando nessuna coppia di stringhe condivide un prefisso comune non vuoto: in questo caso, tranne la radice, tutti i nodi interni hanno un solo figlio.

Un trie T per un insieme S di stringhe può essere utilizzato per implementare un insieme o una mappa le cui chiavi siano le stringhe di S . In particolare, per cercare una stringa X in T è sufficiente scendere dalla radice lungo il percorso individuato dai caratteri di X : se tale percorso esiste e termina in una foglia, allora sappiamo che X è una delle stringhe di S . Ad esempio, nel trie della Figura 13.7, scendendo lungo il percorso corrispondente a "bull" si termina in una foglia. Se, invece, il percorso non esiste oppure esiste ma termina in un nodo interno, allora X non è una delle stringhe di S . Nell'esempio della Figura 13.7, il percorso corrispondente a "bet" non esiste e il percorso di "be" termina in un nodo interno: nessuna delle due parole appartiene all'insieme S .

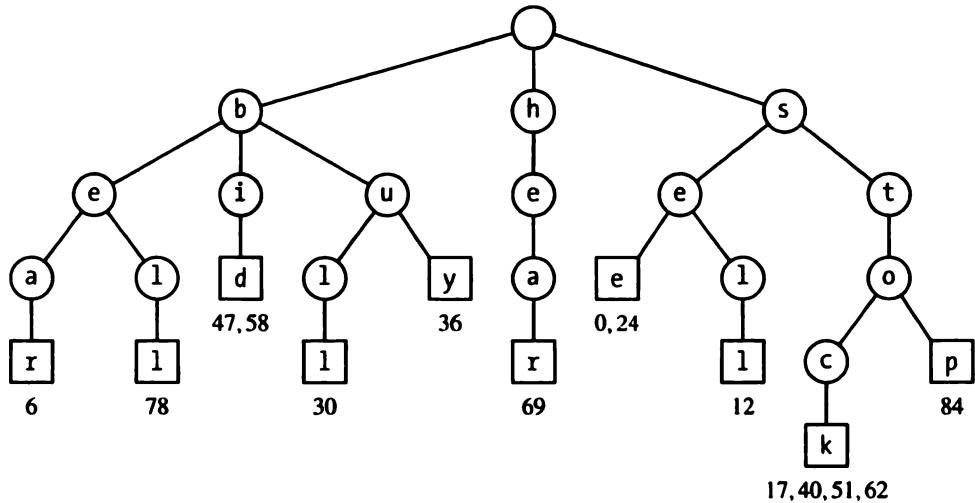
È facile vedere che il tempo d'esecuzione della ricerca di una stringa di lunghezza m è $O(m \cdot |\Sigma|)$, perché si visitano al massimo $m + 1$ nodi di T e in ciascun nodo si impiega un tempo $O(|\Sigma|)$ per determinare quale figlio abbia come etichetta il carattere successivo della stringa che si sta cercando. Si può effettuare quest'ultima ricerca in un tempo limitato da $O(|\Sigma|)$, anche se i figli dei nodi non sono ordinati, perché in ciascun nodo il numero massimo di figli è $|\Sigma|$; questo tempo può essere migliorato e reso $O(\log |\Sigma|)$ o anche $O(1)$, mettendo in corrispondenza i caratteri e i figli mediante una tabella di ricerca o una tabella hash secondaria in ciascun nodo, oppure usando una tabella di ricerca diretta di dimensione $|\Sigma|$, se $|\Sigma|$ è sufficientemente piccolo (come nel caso, ad esempio, di stringhe di DNA). Per queste ragioni, solitamente ci si aspetta che la ricerca di una stringa di lunghezza m venga eseguita in un tempo $O(m)$.

Dalla discussione precedente conseguе che possiamo usare un trie per eseguire una forma speciale di pattern matching, chiamata *word matching*, con il quale si vuole determinare se un dato pattern sia esattamente uguale a una delle parole di un testo. Il *word matching* differisce dal pattern matching standard perché il pattern non può essere una sottostringa

qualsiasi del testo: può essere solamente una delle sue parole. Per risolvere questo problema, ciascuna singola parola del documento originale deve essere aggiunta al trie (come si può vedere nella Figura 13.8). Una semplice estensione di questo schema consente di eseguire anche ricerche di *prefix-matching*, tuttavia non si riescono a eseguire in modo efficiente ricerche di pattern qualsiasi all'interno di un testo (ad esempio, quando il pattern è un suffisso proprio di una parola o si estende su due parole).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
s	e	e	a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
	s	e	e	a	b	u	l	l	?		b	u	y		s	t	o	c	k	!		
46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!		
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88			
h	e	a	r		t	h	e	b	e	l	l	?		s	t	o	p	!				

(a)



(b)

Figura 13.8: Esecuzione di word-matching in un trie standard: (a) testo in cui cercare (vengono escluse la cosiddette *stop word*, cioè articoli e preposizioni, perché porterebbero a frequenti violazioni del vincolo sull'assenza di prefissi all'interno dell'insieme); (b) trie standard per le parole del testo. Nelle foglie del trie sono state aggiunte alcune informazioni: gli indici nel testo in cui inizia la parola rappresentata dal nodo. Ad esempio, la foglia che corrisponde alla parola "stock" contiene un'annotazione che dice che tale parola è presente nel testo a partire dagli indici 17, 40, 51 e 62.

Per costruire un trie standard T per un insieme S di stringhe, possiamo usare un algoritmo incrementale che inserisce una stringa alla volta. Ricordiamo l'ipotesi relativa al fatto che nessuna stringa di S sia un prefisso di un'altra stringa di S . Per inserire una stringa X nel trie T , scendiamo lungo il percorso associato a X in T , partendo dalla radice e , quando rimaniamo bloccati perché non si riesce più a scendere, creiamo una nuova catena di nodi che memorizzino i caratteri rimanenti di X e la inseriamo in modo che il suo nodo iniziale diventi figlio del nodo in cui ci siamo fermati. Il tempo necessario per inserire una parola X di lunghezza m è simile a quello richiesto per la sua ricerca, con caso peggiore $O(m \cdot |\Sigma|)$, oppure valore atteso $O(m)$ se si usa in ciascun nodo una tabella hash secondaria. Di conseguenza, la costruzione dell'intero trie per l'insieme S richiede un tempo atteso $O(n)$, dove n è la lunghezza totale delle stringhe di S .

L'albero trie soffre di una potenziale inefficienza nell'uso dello spazio in memoria e questo ha suggerito lo sviluppo del *trie compresso* (*compressed trie*), nodo anche (per ragioni storiche) come *Patricia trie*. Nello specifico, il problema del trie standard sta nel fatto che possono esserci molti nodi che hanno un solo figlio, la cui esistenza è uno spreco di spazio. Ne parleremo nel prossimo paragrafo.

13.3.2 Trie compresso

Un *trie compresso* (*compressed trie*) è simile al trie standard, ma garantisce che ogni nodo interno del trie abbia almeno due figli. Come si può vedere nella Figura 13.9, questa regola viene messa in atto *comprimendo* in un unico ramo catene di nodi aventi un solo figlio. Sia T un trie standard. Diciamo che un nodo interno v di T è *ridondante* se v non è la radice e ha un solo figlio. Ad esempio, il trie della Figura 13.7 ha otto nodi ridondanti. Inoltre, diciamo che una catena di nodi avente $k \geq 2$ rami, $(v_0, v_1) (v_1, v_2) \dots (v_{k-1}, v_k)$, è *ridondante* se:

- v_i è ridondante per $i = 1, \dots, k - 1$.
- v_0 e v_k non sono ridondanti.

Un trie standard T può essere trasformato in un trie compresso sostituendo ciascuna catena ridondante $(v_0, v_1) (v_1, v_2) \dots (v_{k-1}, v_k)$ costituita da $k \geq 2$ rami con un unico nodo (v_0, v_k) e assegnando a v_k una nuova etichetta che sia la concatenazione delle etichette dei nodi v_1, \dots, v_k .

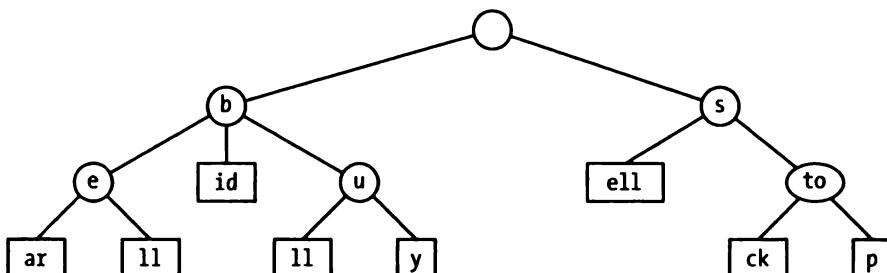


Figura 13.9: Trie compresso per l'insieme di stringhe {bear, bell, bid, bull, buy, sell, stock, stop}, da confrontare con il trie standard per lo stesso insieme, visto nella Figura 13.7. Si osservi che, oltre alla compressione avvenuta nelle foglie, è stato compresso anche un nodo interno, quello con etichetta "to", che è condiviso dalle parole "stock" e "stop".

I nodi compressi, quindi, sono etichettati con stringhe, che sono sottostringhe dell'insieme dato, e non più con singoli caratteri. Il vantaggio di un trie compresso, rispetto a un trie standard, è che il numero di nodi del trie compresso è proporzionale al numero di stringhe e non più alla loro lunghezza totale, come affermato dalla proposizione seguente (da confrontare con la Proposizione 13.4).

Proposizione 13.5: *Un trie compresso T che memorizza una raccolta S di s stringhe i cui caratteri appartengono all'alfabeto Σ di dimensione d gode delle seguenti proprietà:*

- Ogni nodo interno di T ha almeno due figli e al massimo d figli.
- T ha s foglie.
- Il numero di nodi di T è $O(s)$.

Il lettore attento si chiederà se la compressione dei percorsi fornisca effettivamente qualche vantaggio, dal momento che è controbilanciata dal corrispondente allungamento delle etichette dei nodi. In effetti, un trie compresso è davvero vantaggioso soltanto quando viene utilizzato come struttura di indicizzazione *ausiliaria* per un insieme di stringhe già memorizzato in una struttura principale, perché in tal modo non è più necessario memorizzare nel trie tutti i caratteri delle stringhe.

Supponiamo, ad esempio, che l'insieme S di stringhe sia memorizzato in un array, in modo che le stringhe siano $S[0], S[1], \dots, S[s - 1]$. Invece di memorizzare l'etichetta X di un nodo in modo esplicito, la rappresentiamo semplicemente come combinazione di tre numeri interi (i, j, k) , tali che $X = S[i][j..k]$, cioè X è la sottostringa di $S[i]$ costituita dai caratteri che vanno dall'indice j all'indice k , inclusi (si veda un esempio nella Figura 13.10, confrontandolo anche con il trie standard della Figura 13.8).

Questo ulteriore schema di compressione ci consente di ridurre lo spazio totale occupato dal trie, passando da $O(n)$ per il trie standard a $O(s)$ per il trie compresso, essendo n la lunghezza totale delle stringhe di S e s il numero di stringhe di S . Abbiamo ovviamente ancora bisogno di memorizzare da qualche parte le diverse stringhe che appartengono a S , ma abbiamo ridotto lo spazio occupato dal trie.

La ricerca in un trie compresso non è necessariamente più veloce di quella in un trie standard, perché c'è ancora bisogno di confrontare ciascun carattere del pattern cercato con le etichette, potenzialmente costituite da più caratteri, che si incontrano attraversando i percorsi nel trie.

13.3.3 Trie dei suffissi

Una delle applicazioni principali degli alberi trie riguarda il caso in cui le stringhe dell'insieme S sono tutti i suffissi di una data stringa X . Un tale trie è chiamato *trie dei suffissi* (*suffix trie*) o *albero dei suffissi* della stringa X . Ad esempio, la Figura 13.11a mostra il trie dei suffissi per la stringa "minimize", che ha otto suffissi. Nel caso di un trie dei suffissi, la rappresentazione compatta che abbiamo visto nel paragrafo precedente può essere ulteriormente semplificata: l'etichetta di ciascun nodo è una coppia "j..k" e rappresenta la stringa $X[j..k]$ (come si può vedere nella Figura 13.11b). Per soddisfare il vincolo, tipico di tutti i trie, che nessun suffisso di X sia un prefisso di un altro suffisso di X , possiamo aggiungere alla fine di X (e, quindi, alla fine di ciascun suo suffisso) un carattere speciale, spesso indicato

con \$, che non faccia parte dell'alfabeto originario del problema, Σ . Quindi, se la stringa X ha lunghezza n , costruiamo un trie per l'insieme delle n stringhe $X[j..n - 1]\$$, con $j = 0, \dots, n - 1$.

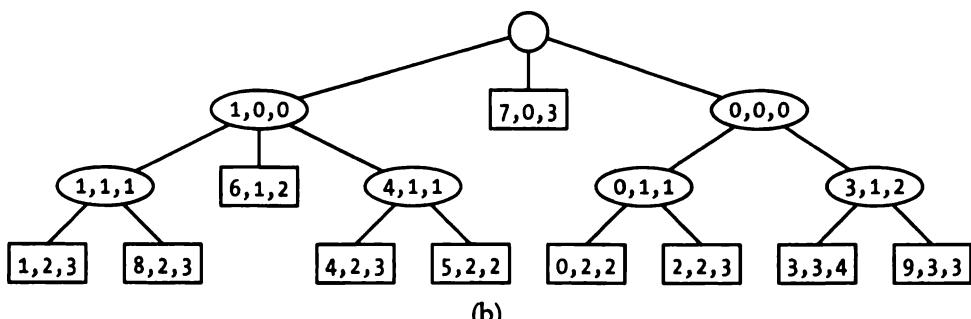
	0 1 2 3 4
$S[0] =$	s e e
$S[1] =$	b e a r
$S[2] =$	s e l l
$S[3] =$	s t o c k

	0 1 2 3
$S[4] =$	b u l l
$S[5] =$	b u y

	0 1 2 3
$S[7] =$	h e a r
$S[8] =$	b e l l

	0 1 2 3
$S[9] =$	s t o p

(a)



(b)

Figura 13.10: (a) Insieme di stringhe memorizzato in un array. (b) Rappresentazione compatta del trie compresso di S .

Risparmiare spazio

L'uso di un trie dei suffissi ci consente di risparmiare spazio rispetto a un trie standard, per via di alcune tecniche di compressione che si possono usare, tra le quali quelle già viste per i trie compressi.

Il vantaggio di una rappresentazione compatta di un trie risulta evidente nel caso dei trie dei suffissi. Dato che la lunghezza totale dei suffissi di una stringa X di lunghezza n è:

$$1 + 2 + \dots + n = \frac{n(n+1)}{2},$$

la memorizzazione in un trie di tutti i suffissi di X in modo esplicito richiederebbe uno spazio $O(n^2)$, mentre il trie dei suffissi rappresenta queste stringhe in modo implicito occupando uno spazio $O(n)$, come asserito in modo formale dalla proposizione che segue.

Proposizione 13.6: *La rappresentazione compatta di un trie dei suffissi T per la stringa X di lunghezza n usa uno spazio $O(n)$.*

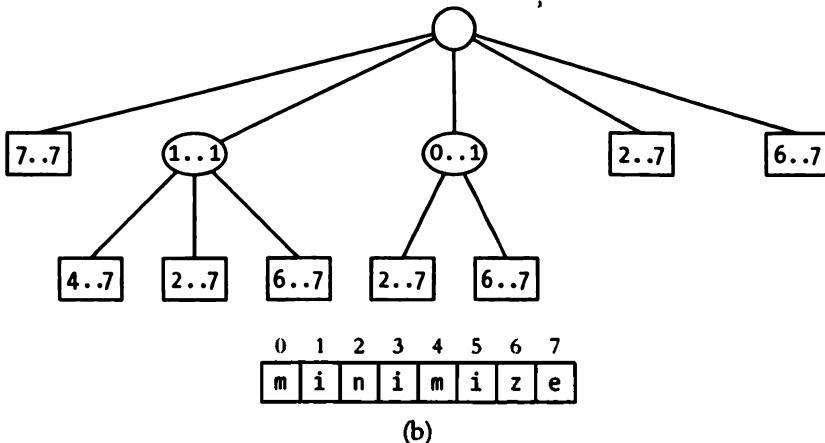
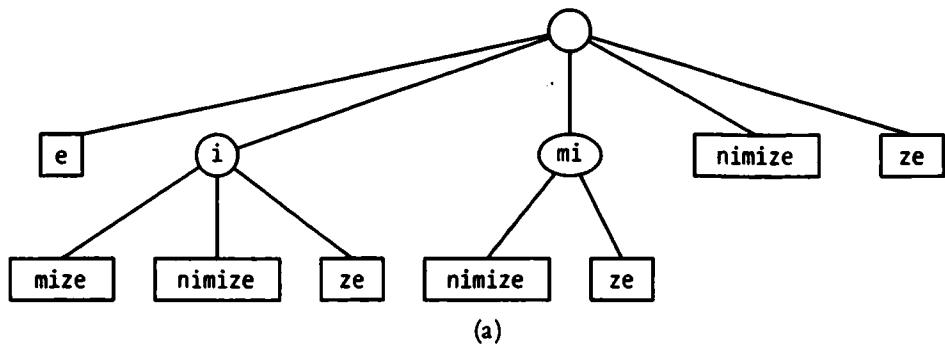


Figura 13.11: (a) Trie dei suffissi T per la stringa $X = \text{"minimize"}$. (b) Rappresentazione compatta di T , dove la coppia $j..k$ rappresenta la sottostringa $X[j..k]$ della stringa di riferimento.

Costruzione

Possiamo costruire il trie dei suffissi per una stringa di lunghezza n usando un algoritmo incrementale come quello visto nel Paragrafo 13.3.1. Questa costruzione richiede un tempo $O(|\Sigma|n^2)$ perché la lunghezza totale dei suffissi è una funzione quadratica di n , però il trie dei suffissi (compatto) per una stringa di lunghezza n può essere costruito in un tempo $O(n)$ usando un algoritmo specifico, diverso da quello usato per i trie standard. Questo algoritmo di costruzione in un tempo lineare è, però, piuttosto complesso e non lo riportiamo qui. Comunque, quando vogliamo usare un trie dei suffissi per risolvere problemi, possiamo basarci sull'esistenza di questo algoritmo di costruzione veloce.

Uso di un trie dei suffissi

Il trie dei suffissi T per una stringa X può essere utilizzato per risolvere in modo efficiente il problema del pattern matching sul testo X : possiamo, infatti, determinare se un pattern P è una sottostringa di X cercando di seguire un percorso associato a P in T , perché P è una sottostringa di X se e solo se è possibile seguire tale percorso. La ricerca all'interno del trie T seguendo un percorso in discesa a partire dalla radice presuppone che i nodi di T memorizzino alcune informazioni aggiuntive, rispetto alla rappresentazione compatta che abbiamo visto per il trie dei suffissi:

Se il nodo v ha etichetta $j..k$ e Y è la stringa di lunghezza y associata al percorso che va dalla radice a v (incluso), allora $X[k - y + 1..k] = Y$.

Questa proprietà garantisce che, se il pattern è presente nel testo, in un tempo $O(m)$ possiamo calcolare l'indice iniziale della sua occorrenza nel testo.

13.3.4 Indicizzazione nei motori di ricerca

Il World Wide Web contiene un'enorme quantità di documenti di testo (le pagine web). Le informazioni relative a queste pagine possono essere raccolte usando un programma chiamato *Web crawler*, che poi memorizza queste informazioni in una base di dati. Un *motore di ricerca* (*search engine*) per il Web permette agli utenti di recuperare da tale base di dati le informazioni più rilevanti, perché è in grado di individuare, tra le pagine del Web, le più rilevanti tra quelle che contengono le parole chiave (*keyword*) specificate. In questo paragrafo presenteremo un modello semplificato di un motore di ricerca.

Indici inversi

L'informazione fondamentale memorizzata da un motore di ricerca è una mappa, detta *indice inverso* (*inverted index*) o *file inverso* (*inverted file*), che contiene coppie chiave–valore del tipo (w, L) , dove w è una parola e L è un contenitore di pagine che contengono la parola w . Le chiavi (cioè le parole) di questa mappa sono chiamate *termini indice* (*index term*) e dovrebbero costituire un insieme di voci di un vocabolario e di nomi propri, con la maggiore dimensione possibile. I valori di questa mappa sono detti *liste di occorrenza* (*occurrence list*) e dovrebbero contenere il massimo numero possibile di pagine web.

Un indice inverso può essere facilmente implementato con una struttura dati di questo tipo:

1. Un array che memorizza le liste di occorrenza associate ai singoli termini indice (senza rispettare alcun ordinamento particolare).
2. Un trie compresso associato all'insieme dei termini indice (cioè delle chiavi), con le foglie che memorizzano il valore dell'indice in corrispondenza del quale si trova, nell'array, la lista di occorrenza associata al termine indice.

Le liste di occorrenza vengono memorizzate al di fuori del trie per fare in modo che la dimensione della struttura dati che realizza il trie sia sufficientemente piccola da trovar posto nella memoria interna, mentre, per via della grande dimensione totale che hanno, le liste di occorrenza devono essere memorizzate sul disco, nella memoria di massa.

Con questa struttura dati, la ricerca di una singola parola chiave è simile a un problema di *word matching* (visto nel Paragrafo 13.3.1): si cerca la parola nel trie e si restituisce la lista di occorrenza associata.

Quando l'utente fornisce più parole chiave e desidera ottenere un elenco delle pagine che contengono *tutte* le parole chiave date, si recupera la lista di occorrenza di ciascuna parola usando il trie e si restituisce la loro intersezione. Per agevolare il calcolo dell'intersezione, le liste di occorrenza devono essere implementate con una sequenza ordinata in base all'indirizzo della pagina o con una mappa, in modo da consentire un'esecuzione efficiente delle operazioni tra insiemi.

Oltre a risolvere il problema basilare della restituzione di un elenco di pagine contenenti le parole chiave cercate, i motori di ricerca forniscono anche un ulteriore servizio, assai importante: restituiscono le pagine con un *punteggio (ranking)* associato, in base alla loro rilevanza. La progettazione di algoritmi veloci e accurati per assegnare un punteggio alle pagine è uno dei problemi cruciali nel mondo dei motori di ricerca e costituisce una delle sfide principali per i ricercatori informatici e per le aziende di commercio elettronico.

13.4 Compressione del testo e metodo *greedy*

In questo paragrafo tratteremo l'importante tema della *compressione del testo*: in questo problema ci viene assegnata una stringa X definita in un determinato alfabeto, come l'insieme dei caratteri ASCII o Unicode, e vogliamo codificare in modo efficiente X mediante una stringa binaria Y di dimensione minore, usando soltanto i caratteri 0 e 1. La compressione del testo è utile in molte situazioni, ad esempio per ridurre la banda richiesta per la trasmissione digitale del testo stesso e per minimizzare il tempo di trasmissione. Analogamente, la compressione del testo è utile per memorizzare in modo più efficiente i documenti di grandi dimensioni, consentendo a un dispositivo di memorizzazione con capacità prefissata di contenere un maggior numero di documenti.

Il metodo per la compressione del testo che vedremo in questo paragrafo è la *codifica di Huffman*. Gli schemi standard di codifica del testo, come ASCII, usano stringhe binarie di lunghezza fissa per codificare i singoli caratteri (con 7 o 8 bit per ciascun carattere nella codifica ASCII tradizionale o, rispettivamente, estesa). Il sistema Unicode fu originariamente proposto come rappresentazione di lunghezza fissa a 16 bit, anche se le codifiche più diffuse riducono l'occupazione di spazio consentendo di utilizzare un minor numero di bit per alcuni gruppi di caratteri utilizzati molto frequentemente, come quelli appartenenti all'insieme ASCII. La codifica di Huffman risparmia spazio rispetto a una codifica a lunghezza fissa, perché usa stringhe (dette *code-word*, cioè parole di codice) più corte per codificare i caratteri più ricorrenti (detti *ad alta frequenza*) e stringhe più lunghe per codificare i caratteri meno ricorrenti (cioè *a bassa frequenza*). Inoltre, la codifica di Huffman usa codici di lunghezza variabile specificatamente ottimizzati per una data stringa X i cui caratteri appartengono a un alfabeto qualunque. L'ottimizzazione è basata sull'uso delle *frequenze* dei caratteri: per ogni carattere c , viene usato un conteggio $f(c)$ del numero di occorrenze di c nella stringa X .

Per codificare la stringa X , convertiamo ciascun suo carattere in una parola di codice (*code-word*) di lunghezza variabile e concateniamo tutte le parole di codice così ottenute, generando la codifica Y di X . Al fine di evitare ambiguità, esigiamo che, nello schema di codifica adottata, non sia presente nessuna parola di codice che sia un prefisso di un'altra parola di codice nello stesso schema di codifica. Uno schema di codifica di questo tipo è chiamato *codifica a prefisso* e semplifica la decodifica di Y , necessaria per recuperare X (si veda la Figura 13.12). Anche con questo vincolo restrittivo, il risparmio di spazio ottenibile con una codifica a prefisso e di lunghezza variabile può essere significativo, in particolar modo se è presente una grande variabilità nelle frequenze dei caratteri (come avviene nei testi scritti nella maggior parte dei linguaggi naturali).

L'algoritmo di Huffman che genera una codifica ottimale, a prefisso e di lunghezza variabile per la stringa X è basato sulla costruzione di un albero binario T che rappresenta

lo schema di codifica. Ogni ramo di T rappresenta un bit in una parola di codice, con i rami che vanno verso un figlio sinistro che rappresentano un carattere "0" e i rami che vanno verso un figlio destro che rappresentano un carattere "1". Ogni foglia v è associata a uno specifico carattere e la *code-word* che codifica quel carattere è definita dalla sequenza di bit associati ai rami del percorso che scende dalla radice di T a v (si veda la Figura 13.12). Ogni foglia v ha una *frequenza*, $f(v)$, che è semplicemente la frequenza in X del carattere associato a v . Inoltre, assegniamo una frequenza $f(v)$ anche a tutti i nodi interni v di T : la somma delle frequenze delle foglie appartenenti al sottoalbero che ha radice in v .

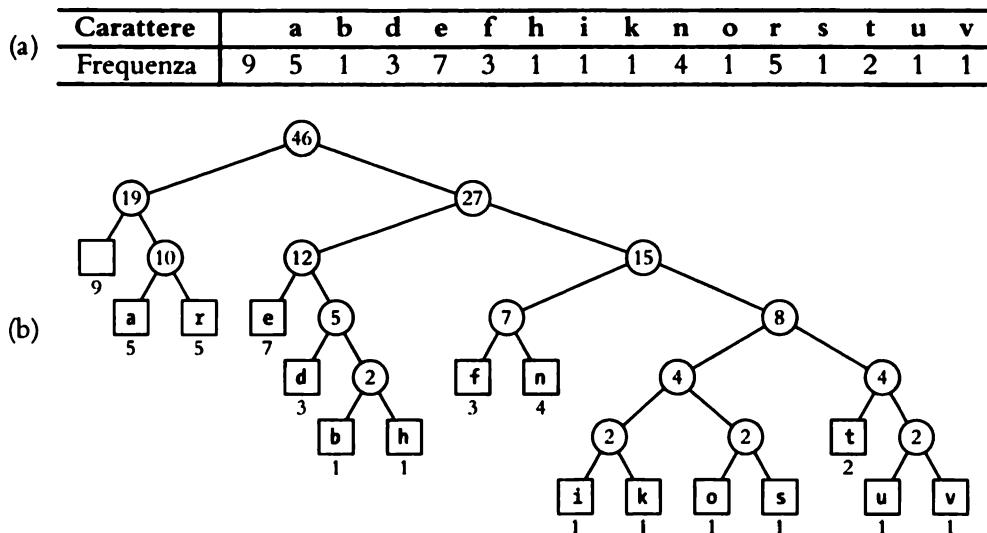


Figura 13.12: Un esempio di codifica di Huffman applicata alla stringa $X = \text{"a fast runner need never be afraid of the dark"}$: (a) frequenza di ciascun carattere di X ; (b) albero di Huffman T per la stringa X . La codifica del carattere c si ottiene seguendo il percorso che scende dalla radice di T fino alla foglia in cui è memorizzato c , associando un figlio sinistro alla cifra 0 e un figlio destro alla cifra 1. Ad esempio, la parola di codice per "r" è 011, mentre quella per "h" è 10111.

13.4.1 L'algoritmo di codifica di Huffman

L'algoritmo di codifica di Huffman inizia creando, per ognuno dei d caratteri distinti presenti nella stringa da codificare X , un albero binario avente la sola radice, che, quindi, è anche una foglia e contiene il carattere in esame. L'algoritmo procede per fasi: in ciascuna fase, prende i due alberi binari aventi le frequenze più basse e li fonde in un unico albero binario, ripetendo la procedura finché non rimane un unico albero (come descritto nel Codice 13.5).

Nell'algoritmo di Huffman, ciascuna iterazione del ciclo `while` può essere implementata in un tempo $O(\log d)$, usando una coda prioritaria Q realizzata mediante uno heap. Inoltre, ogni iterazione estrae due nodi da Q e ve ne aggiunge uno, una procedura che verrà ripetuta $d - 1$ volte prima che in Q rimanga uno e un solo nodo. Quindi, il tempo d'esecuzione di questo algoritmo è $O(n + d \log d)$. Sebbene la dimostrazione della correttezza di questo algoritmo vada oltre gli obiettivi del testo, osserviamo che l'intuizione su cui si basa deriva

da un'idea semplice: qualunque schema di codifica ottimale può essere convertito in un altro schema di codifica ottimale nel quale le parole di codice relative ai due caratteri, a e b , aventi frequenza minima differiscono soltanto nel loro ultimo bit. Ripetendo questo ragionamento per una stringa, sostituendo di volta in volta a e b con caratteri effettivi, si ottiene la seguente proprietà.

Proposizione 13.7: *L'algoritmo di Huffman costruisce in un tempo $O(n + d \log d)$ uno schema di codifica ottimale a prefissi per una stringa di lunghezza n contenente d caratteri distinti.*

Codice 13.5: L'algoritmo di codifica di Huffman.

Algoritmo Huffman(X):

```

Input: Una stringa  $X$  di lunghezza  $n$  contenente  $d$  caratteri distinti
Output: L'albero di codifica per  $X$ 
calcola la frequenza  $f(c)$  di ogni carattere  $c$  di  $X$ 
inizializza una coda prioritaria  $Q$ 
for ogni carattere  $c$  in  $X$  do
    crea un albero binario  $T$  avente solo la radice, che contiene  $c$ 
    inserisci  $T$  in  $Q$  con chiave  $f(c)$ 
while  $Q.size() > 1$  do
    sia  $e_1 = Q.removeMin()$  la voce che ha chiave  $f_1$  e valore  $T_1$ 
    sia  $e_2 = Q.removeMin()$  la voce che ha chiave  $f_2$  e valore  $T_2$ 
    crea un nuovo albero binario  $T$  con  $T_1$  e  $T_2$  come sottoalberi sinistro e destro
    inserisci  $T$  in  $Q$  con chiave  $f_1 + f_2$ 
sia  $e = Q.removeMin()$  la voce che ha valore  $T$ 
return l'albero  $T$ 
```

13.4.2 Il metodo greedy

L'algoritmo di Huffman, che costruisce uno schema di codifica ottimale, è un esempio di applicazione di uno schema di progettazione di algoritmi che prende il nome di *metodo greedy* (cioè “goloso”). Questo schema di progettazione si applica, in generale, a problemi di ottimizzazione, dove si cerca di costruire una qualche struttura minimizzandone o massimizzandone alcune proprietà.

La formulazione generale del metodo di progettazione *greedy* è semplice forse quanto il metodo a forza bruta. Per risolvere un dato problema di ottimizzazione usando il metodo *greedy* si procede mediante una sequenza di scelte. La sequenza inizia da una qualche ben nota condizione di partenza e calcola il “costo” di tale condizione iniziale. Lo schema, poi, chiede iterativamente di compiere una scelta dopo l'altra, identificando ogni volta la decisione che consente di ottenere, tra i miglioramenti di costo garantiti da ciascuna scelta ammissibile, quello migliore. Non sempre, però, questo approccio porta a una soluzione ottimale.

Ma ci sono molto problemi per i quali lo fa: sono quei problemi che sono dotati di una qualche proprietà di *scelta greedy*. Si tratta di una proprietà secondo cui una condizione ottima globale può essere raggiunta mediante una serie di scelte localmente ottime (ciascuna delle quali, cioè, rappresenta la scelta migliore tra quelle ammissibili nel momento della

scelta), a partire da una condizione ben definita. Il problema del calcolo di uno schema di codifica ottimale, a prefissi e con parole di codice di lunghezza variabile, è soltanto un esempio di problema che possiede la proprietà di scelta greedy.

13.5 Programmazione dinamica

In questo paragrafo parleremo dello schema di progettazione di algoritmi detto *programmazione dinamica (dynamic programming)*. Questa tecnica è simile alla strategia dividi-e-conquista, vista nel Paragrafo 12.1.1, per il fatto che si può applicare a una grande varietà di problemi diversi. Spesso la programmazione dinamica può essere utilizzata per generare algoritmi che richiedono un tempo d'esecuzione polinomiale per risolvere problemi che sembrerebbero richiedere un tempo esponenziale; inoltre, gli algoritmi prodotti dall'applicazione della tecnica di programmazione dinamica sono solitamente piuttosto semplici, spesso realizzabili con poche linee di codice che descrivono alcuni cicli annidati che riempiono una tabella.

13.5.1 Moltiplicazione matriciale a catena

Invece di partire da una spiegazione delle componenti generali della tecnica di programmazione dinamica, cominciamo con un esempio concreto, per risolvere un problema classico. Supponiamo di voler calcolare il prodotto matematico di un insieme di n matrici bidimensionali:

$$A = A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1},$$

dove A_i è una matrice $d_i \times d_{i+1}$, con $i = 0, 1, 2, \dots, n - 1$. Nell'algoritmo standard di moltiplicazione tra matrici (che è quello che useremo), per moltiplicare una matrice B , avente dimensioni $d \times e$, per una matrice C , avente dimensioni $e \times f$, si calcola la matrice prodotto, A , avente dimensioni $d \times f$, in questo modo:

$$A[i][j] = \sum_{k=0}^{e-1} B[i][k] \cdot C[k][j].$$

Questa definizione implica che la moltiplicazione tra matrici è associativa, cioè implica che $B \cdot (C \cdot D) = (B \cdot C) \cdot D$. Quindi, nel calcolo di A come prodotto di n matrici, possiamo mettere le parentesi dove preferiamo: il risultato ottenuto sarà sempre lo stesso, anche se non eseguiremo necessariamente sempre lo stesso numero di moltiplicazioni primitive (cioè scalari, tra coppie di numeri), in relazione al posizionamento delle parentesi, come si evince dall'esempio seguente.

Esempio 13.8: Sia B una matrice 2×10 , C una matrice 10×50 e D una matrice 50×20 . Il calcolo di $B \cdot (C \cdot D)$ richiede $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10400$ moltiplicazioni, mentre il calcolo di $(B \cdot C) \cdot D$ richiede soltanto $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ moltiplicazioni.

Il problema della *moltiplicazione matriciale a catena* consiste nel determinare, nell'espressione che definisce il prodotto A di n matrici, il posizionamento delle parentesi in modo

da minimizzare il numero totale di moltiplicazioni scalari che devono essere eseguite. Come evidenziato dall'esempio precedente, le differenze indotte dal posizionamento delle parentesi possono essere molto rilevanti, per cui l'individuazione di una buona soluzione può produrre risparmi di tempo significativi.

Definizione di sotto-problemi

Una possibile soluzione del problema della moltiplicazione matriciale a catena consiste nella semplice elencazione di tutti i possibili posizionamenti delle parentesi nell'espressione che definisce il prodotto A , per poi determinare il numero di moltiplicazioni scalari eseguite in ciascun caso. Sfortunatamente, l'insieme di tutti i diversi posizionamenti possibili delle parentesi nell'espressione A che moltiplica n matrici ha la stessa dimensione dell'insieme di tutti i diversi alberi binari che hanno n foglie, e questa dimensione è un numero esponenziale in funzione di n . Quindi, questo approccio algoritmico che usa la forza bruta viene eseguito in un tempo esponenziale, perché il numero di possibili posizionamenti delle parentesi in un'espressione aritmetica che gode della proprietà associative è esponenziale.

Possiamo, però, migliorare in modo significativo le prestazioni ottenute dall'algoritmo a forza bruta sfruttando alcune osservazioni relative alla natura del problema del prodotto matriciale concatenato. La prima di queste osservazioni riguarda il fatto che il problema può essere suddiviso in *sotto-problemi*. In questo caso, possiamo definire un certo numero di sotto-problemi diversi, ciascuno dei quali deve calcolare il posizionamento migliore delle parentesi in una sotto-espressione come $A_i \cdot A_{i+1} \cdots A_j$. Per usare una notazione sintetica, indichiamo con $N_{i,j}$ il numero minimo di moltiplicazioni necessarie per calcolare questa sotto-espressione. Quindi, il problema originario di moltiplicazione matriciale a catena può essere ricondotto al calcolo di $N_{0,n-1}$. Questa osservazione è importante, ma per poter applicare la tecnica di programmazione dinamica ce ne serve un'altra.

Definizione di soluzioni ottimali

L'altra osservazione importante che possiamo fare in merito al problema della moltiplicazione matriciale a catena riguarda il fatto che sia possibile definire una soluzione ottimale per un particolare sotto-problema in termini delle soluzioni ottimali dei suoi sotto-problemi, una proprietà che chiamiamo *ottimalità dei sotto-problemi*.

Nel caso del problema della moltiplicazione matriciale a catena, osserviamo che, indipendentemente dal posizionamento delle parentesi in un sotto-problema, l'operazione complessiva deve concludersi con una moltiplicazione tra due matrici, cioè il posizionamento delle parentesi nella sotto-espressione $A_i \cdot A_{i+1} \cdots A_j$ deve avere la forma $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, per un qualche valore di $k \in \{i, i+1, \dots, j-1\}$. Inoltre, qualunque sia il valore di k corretto, anche i prodotti parziali $(A_i \cdots A_k)$ e $(A_{k+1} \cdots A_j)$ devono essere risolti in modo ottimale. Se così non fosse, esisterebbe una soluzione globalmente ottimale in cui uno di questi sotto-problemi sarebbe risolto in modo non ottimale, ma questo è impossibile, perché allora potremmo ridurre ulteriormente il numero di moltiplicazioni complessivo sostituendo la soluzione del sotto-problema non ottimale con una sua soluzione ottimale. Questa osservazione implica che si possa definire esplicitamente il problema di ottimizzazione avente per obiettivo $N_{i,j}$ in termini delle soluzioni ottimali di altri sotto-problemi: possiamo calcolare $N_{i,j}$ considerando ognuno dei possibili valori di k in corrispondenza del quale avverrà la moltiplicazione finale e prendendo il valore minimo tra tutte queste situazioni.

Progettazione di un algoritmo mediante programmazione dinamica

Possiamo quindi caratterizzare in questo modo la soluzione ottimale del sotto-problema che riguarda il calcolo di $N_{i,j}$:

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\},$$

dove $N_{i,i} = 0$, perché quando la matrice è una sola non c'è alcuna moltiplicazione scalare da eseguire. Quindi, $N_{i,j}$ si ottiene prendendo il valore minimo tra le somme che hanno come addendi il numero di moltiplicazioni scalari richieste per calcolare le due sotto-espressioni e il numero di moltiplicazioni richieste per eseguire la moltiplicazione matriciale conclusiva, considerando tutti i possibili posizionamenti di tale moltiplicazione conclusiva.

Osserviamo che è presente una *condizione di sotto-problemi* che ci impedisce di suddividere il problema originario in sotto-problemi completamente indipendenti (come avremmo fatto se avessimo applicato la strategia dividi-e-conquista). Possiamo, però, utilizzare l'equazione che descrive $N_{i,j}$ per derivare un algoritmo efficiente per calcolare i valori $N_{i,j}$ in una modalità *bottom-up*, "dal basso verso l'alto", memorizzando le soluzioni intermedie in una tabella contenente tutti i valori $N_{i,j}$. Possiamo iniziare semplicemente con le assegnazioni $N_{i,i} = 0$ per $i = 0, 1, \dots, n - 1$. Poi, applichiamo l'equazione generale che definisce $N_{i,j}$ per calcolare i valori $N_{i,i+1}$, perché questi dipendono soltanto dai valori $N_{i,i}$ e $N_{i+1,i+1}$, che sono disponibili (per quanto avvenuto al passo precedente). Noti che siano i valori $N_{i,i+1}$, possiamo poi procedere calcolando i valori $N_{i,i+2}$, e così via. Quindi, possiamo calcolare i valori $N_{i,j}$ a partire da valori calcolati in precedenza, fino a calcolare il valore finale, $N_{0,n-1}$, che è il numero che stiamo cercando. Il Codice 13.6 presenta un'implementazione in Java di questa soluzione mediante *programmazione dinamica*, usando le tecniche viste nel Paragrafo 3.1.5 per lavorare in Java con un array bidimensionale.

Codice 13.6: Algoritmo per la soluzione del problema della moltiplicazione matriciale a catena utilizzando la programmazione dinamica.

```

1  public static int[][] matrixChain(int[] d) {
2      int n = d.length - 1;           // numero di matrici
3      int[][] N = new int[n][n];     // matrice n x n inizializzata con zeri
4      for (int b=1; b < n; b++)    // numero di prodotti in una sotto-catena
5          for (int i=0; i < n - b; i++) { // inizio di una sotto-catena
6              int j = i + b;           // fine della sotto-catena
7              N[i][j] = Integer.MAX_VALUE; // usato come "infinito"
8              for (int k=i; k < j; k++)
9                  N[i][j] = Math.min(N[i][j], N[i][k] + N[k+1][j] + d[i]*d[k+1]*d[j+1]);
10         }
11     return N;
12 }
```

Siamo, quindi, in grado di calcolare $N_{0,n-1}$ con un algoritmo che è costituito, in pratica, da tre cicli annidati (il terzo dei quali calcola i termini minimi). Ciascuno di questi cicli effettua al massimo n iterazioni per ogni sua esecuzione, con una quantità di lavoro aggiuntivo svolto al proprio interno, quindi il tempo totale richiesto per l'esecuzione di questo algoritmo è $O(n^3)$.

13.5.2 DNA e allineamento di sequenze di caratteri

Un problema di elaborazione di testi molto diffuso, che emerge nella genetica e nell'ingegneria del software, consiste nella verifica di somiglianza tra due stringhe di testo. In un'applicazione genetica, le due stringhe potrebbero corrispondere a due porzioni di DNA, tra le quali si vogliono evidenziare eventuali somiglianze. Analogamente, in un'applicazione di ingegneria del software, le due stringhe potrebbero essere due versioni del codice sorgente di uno stesso programma, per le quali si vogliono determinare le modifiche apportate per passare da una versione all'altra. In effetti, l'individuazione di somiglianze tra due stringhe è un problema tanto frequente che i sistemi operativi Unix e Linux mettono a disposizione proprio un programma, che si chiama `diff`, che confronta due file di testo.

Data una stringa $X = x_0x_1x_2 \dots x_{n-1}$, una *sotto-sequenza* (*subsequence*) di X è una qualsiasi stringa avente la forma $x_{i_1}x_{i_2} \dots x_{i_k}$, con $i_j < i_{j+1}$, quindi è una sequenza di caratteri che appartengono a X ma non vi compaiono necessariamente in posizioni consecutive, pur comparendovi nello stesso ordine. Ad esempio, la stringa *AAAG* è una sotto-sequenza della stringa *CGATAATTGAGA*.

Il problema di cui parleremo, relativo a DNA e a somiglianza tra testi, è detto **LCS**, *longest common subsequence*, cioè, date due stringhe, consiste nella ricerca della *sotto-sequenza comune più lunga*. Indichiamo con $X = x_0x_1x_2 \dots x_{n-1}$ e $Y = y_0y_1y_2 \dots y_{m-1}$ due stringhe, definite in un alfabeto (come l'alfabeto $\{A, C, G, T\}$ solitamente utilizzato nella genomica computazionale), per le quali ci viene chiesto di individuare la più lunga stringa S che sia sotto-sequenza tanto di X quanto di Y . Una strategia che consente di risolvere questo problema consiste nell'elencare tutte le sotto-sequenze di X e, poi, prendere quella più lunga che sia anche una sotto-sequenza di Y . Dato che ciascun carattere di X può appartenere oppure no a ciascuna diversa sotto-sequenza, esistono potenzialmente 2^n diverse sotto-sequenze di X , ciascuna delle quali necessita di un tempo $O(m)$ perché si possa determinare se è anche una sotto-sequenza di Y . Così, questo approccio a forza bruta dà luogo a un algoritmo che richiede un tempo esponenziale, $O(2^n m)$, veramente inefficiente. Fortunatamente, il problema LCS è risolvibile in modo efficiente usando la *programmazione dinamica*.

Le componenti di una soluzione che usa la programmazione dinamica

Come già detto, la tecnica di programmazione dinamica viene usata principalmente nei problemi di *ottimizzazione*, dove si vuole trovare il modo "migliore" per fare una determinata cosa.

Sotto-problemi semplici: Deve esistere un modo per scomporre ripetutamente in sotto-problemi il problema di ottimizzazione complessivo. Inoltre, ci deve essere un modo per rendere parametrici i sotto-problemi usando un piccolo numero di indici, come i, j, k e così via.

Ottimizzazione dei sotto-problemi: Una soluzione ottimale del problema globale deve essere ottenibile componendo le soluzioni ottimali di sotto-problemi.

Sovrapposizione dei sotto-problemi: Le soluzioni ottimali di sotto-problemi tra loro non correlati possono avere sotto-problemi in comune.

Applicazione della programmazione dinamica al problema LCS

Ricordiamo che, nel problema LCS, ci sono date due stringhe di caratteri, X e Y , di lunghezza, rispettivamente, n e m , e ci viene chiesto di trovare una stringa S di lunghezza massima che sia sotto-sequenza tanto di X quanto di Y . Dato che X e Y sono stringhe di caratteri, disponiamo di un insieme naturale di indici con cui definire sotto-problemi: gli indici dei caratteri nelle stringhe X e Y . Definiamo, quindi, il sotto-problema relativo al calcolo del valore $L_{j,k}$, simbolo che useremo per indicare la lunghezza della più lunga stringa che sia una sotto-sequenza tanto dei primi j caratteri di X quanto dei primi k caratteri di Y , cioè sia sotto-sequenza dei prefissi $X[0..j-1]$ e $Y[0..k-1]$. Se $j = 0$ oppure $k = 0$, allora il valore di $L_{j,k}$ è banalmente 0.

Quando $j \geq 1$ e $k \geq 1$, questa definizione ci consente di scrivere ricorsivamente $L_{j,k}$ in termini delle soluzioni ottimali dei sotto-problemi. Questa definizione dipende dal caso in cui ci troviamo (si veda la Figura 13.13).

- $x_{j-1} = y_{k-1}$. In questo caso, l'ultimo carattere di $X[0..j-1]$ è uguale all'ultimo carattere di $Y[0..k-1]$. Affermiamo e dimostriamo che questo carattere comune appartiene alla più lunga sotto-sequenza comune ai prefissi $X[0..j-1]$ e $Y[0..k-1]$. Supponiamo, per assurdo, che questo non sia vero. Allora deve esistere una sotto-sequenza comune di lunghezza massima $x_{a_1}x_{a_2}\dots x_{a_t} = y_{b_1}y_{b_2}\dots y_{b_t}$. Se $x_{a_t} = x_{j-1}$ oppure $y_{b_t} = y_{k-1}$, otteniamo la stessa sequenza ponendo $a_i = j-1$ e $b_i = k-1$. Se, invece, $x_{a_t} \neq x_{j-1}$ e $y_{b_t} \neq y_{k-1}$, allora otteniamo una sequenza comune ancora più lunga aggiungendo il carattere $x_{j-1} = y_{k-1}$ alla fine. Di conseguenza, una sotto-sequenza comune a $X[0..j-1]$ e $Y[0..k-1]$ che abbia lunghezza massima deve terminare con il carattere x_{j-1} . Quindi, se $x_{j-1} = y_{k-1}$, poniamo $L_{j,k} = 1 + L_{j-1,k-1}$.
- $x_{j-1} \neq y_{k-1}$. In questo caso, non può esistere una sotto-sequenza comune che contenga sia x_{j-1} sia y_{k-1} . Perciò può esistere una sotto-sequenza comune che termina con x_{j-1} oppure che termina con y_{k-1} (oppure che termina con un carattere diverso da entrambi), ma certamente non una che termina con entrambi! Quindi, se $x_{j-1} \neq y_{k-1}$, poniamo $L_{j,k} = \max\{L_{j-1,k}, L_{j,k-1}\}$.

$X = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ G & T & T & C & C & T & A & A & T & A \end{matrix}$
 $Y = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ C & G & A & T & A & A & T & T & G & A & G \end{matrix}$

 $L_{10,12} = 1 + L_{9,11}$

(a)

$X = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ G & T & T & C & C & T & A & A & T \end{matrix}$
 $Y = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ C & G & A & T & A & A & T & T & G & A & G \end{matrix}$

 $L_{9,11} = \max(L_{9,10}, L_{8,11})$

(b)

Figura 13.13: I due casi che si possono presentare, durante l'esecuzione dell'algoritmo che risolve il problema LCS, per calcolare il valore $L_{j,k}$ quando $j \geq 1$ e $k \geq 1$: (a) $x_{j-1} = y_{k-1}$; (b) $x_{j-1} \neq y_{k-1}$.

L'algoritmo LCS

La definizione di $L_{j,k}$ soddisfa la condizione di l'ottimizzazione dei sotto-problemi, perché non è possibile che una soluzione ottimale del problema globale non derivi dalla composizione delle soluzioni ottimali per i sotto-problemi. Ancora, questa definizione usa la

sovraposizione dei sotto-problemi, perché la soluzione di un sotto-problema, $L_{j,k}$, viene utilizzata per la soluzione di diversi altri sotto-problemi (nello specifico, dei problemi $L_{j+1,k}$, $L_{j,k+1}$ e $L_{j+1,k+1}$). La trasformazione in algoritmo di questa definizione di $L_{j,k}$ è piuttosto banale. Creiamo un array bidimensionale L , con dimensioni $(n + 1) \cdot (m + 1)$, definito per $0 \leq j \leq n$ e $0 \leq k \leq m$. Inizializziamo tutte le sue celle a 0, così, in particolare, tutti i valori del tipo $L_{j,0}$ e $L_{0,k}$ sono 0. Poi, costruiamo iterativamente i valori di L fino a quando non viene calcolato $L_{n,m}$, che è la lunghezza massima di una sotto-seguenza comune delle stringhe X e Y . Il Codice 13.7 presenta un'implementazione in Java di questo algoritmo.

Codice 13.7: Algoritmo per la soluzione del problema LCS utilizzando la programmazione dinamica.

```

1  /** Restituisce la tabella L[j][k] con la lunghezza LCS tra X[0..j-1] e Y[0..k-1]. */
2  public static int[][] LCS(char[] X, char[] Y) {
3      int n = X.length;
4      int m = Y.length;
5      int[][] L = new int[n+1][m+1];
6      for (int j=0; j < n; j++)
7          for (int k=0; k < m; k++)
8              if (X[j] == Y[k])           // allineamento di questa corrispondenza
9                  L[j+1][k+1] = L[j][k] + 1;
10             else                   // decide di ignorare un carattere
11                 L[j+1][k+1] = Math.max(L[j][k+1], L[j+1][k]);
12     return L;
13 }
```

Il tempo d'esecuzione dell'algoritmo LCS è semplice da analizzare perché è dominato dai due cicli `for` annidati, con il ciclo esterno che viene eseguito n volte e il ciclo interno che itera m volte. Dato che l'enunciato condizionale e l'enunciato di assegnazione presenti all'interno del corpo del ciclo richiedono un numero $O(1)$ di operazioni fondamentali, questo algoritmo viene eseguito in un tempo $O(nm)$. Quindi, si può applicare la tecnica di programmazione dinamica al problema LCS migliorandone in modo significativo le prestazioni, rispetto alla soluzione esponenziale ottenibile con un approccio a forza bruta.

Il metodo `LCS` presentato nel Codice 13.7 calcola la lunghezza ($L_{n,m}$) della più lunga sotto-seguenza comune, ma non la sotto-seguenza stessa. Fortunatamente, data la tabella completa dei valori $L_{j,k}$ calcolata dal metodo `LCS`, è facile generare una sotto-seguenza comune avente tale lunghezza massima. La soluzione può essere ricostruita a ritroso, sfruttando il metodo di calcolo della lunghezza $L_{n,m}$. In ogni posizione $L_{j,k}$ della tabella, se $x_j = y_k$, la lunghezza si basa su quella della sotto-seguenza comune più lunga, associata alla lunghezza $L_{j-1,k-1}$, seguita dal carattere comune x_j . Possiamo, quindi, memorizzare il carattere x_j come facente parte della sequenza, per poi continuare l'analisi a partire da $L_{j-1,k-1}$. Se, invece, $x_j \neq y_k$, allora possiamo passare al valore maggiore tra $L_{j-1,k}$ e $L_{j,k-1}$. Proseguiamo con la procedura finché non raggiungiamo un valore del tipo $L_{j,k} = 0$ (se j o k valgono 0 si tratta di un caso limite). Il Codice 13.8 riporta un'implementazione di tale strategia in Java. Il metodo ricostruisce una delle possibili sotto-seguenze comuni di lunghezza massima in un tempo $O(n + m)$, dal momento che ogni passo del ciclo `while` decrementa di un'unità l'indice j o l'indice k (o entrambi). La Figura 13.14 mostra l'algoritmo in azione per calcolare una sotto-seguenza comune di lunghezza massima.

Codice 13.8: Ricostruzione della più lunga sotto-sequenza comune.

```

1  /** Restituisce la più lunga sotto-sequenza comune a X e Y, nota la tabella LCS. */
2  public static char[] reconstructLCS(char[] X, char[] Y, int[][] L) {
3      StringBuilder solution = new StringBuilder();
4      int j = X.length;
5      int k = Y.length;
6      while (L[j][k] > 0)          // rimangono ancora caratteri comuni
7          if (X[j-1] == Y[k-1]) {
8              solution.append(X[j-1]);
9              j--;
10             k--;
11         } else if (L[j-1][k] >= L[j][k-1])
12             j--;
13         else
14             k--;
15     // restituisce, sotto forma di array di char, la versione invertita di solution
16     return solution.reverse().toString().toCharArray();
17 }
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2	2	2	2	2
3	0	0	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	3	3	3	3	3
6	0	1	1	1	2	2	2	3	4	4	4	4	4
7	0	1	1	2	2	3	3	3	4	4	5	5	5
8	0	1	1	2	2	3	4	4	4	4	5	5	6
9	0	1	1	2	3	3	4	5	5	5	5	5	6
10	0	1	1	2	3	4	4	5	5	5	6	6	6

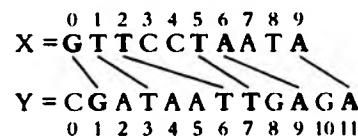


Figura 13.14: Visualizzazione del funzionamento dell'algoritmo che costruisce una sotto-sequenza comune avente lunghezza massima, a partire dall'array L . Un passo in diagonale del percorso evidenziato rappresenta l'utilizzo di un carattere comune (nelle intestazioni di riga e di colonna sono, corrispondentemente, evidenziati gli indici relativi a tale carattere nelle due sequenze).

13.6 Esercizi

Riepilogo e approfondimento

- R-13.1 Elencare i prefissi della stringa $P = "aabbaaa"$ che sono anche suffissi di P .
- R-13.2 Qual è il più lungo prefisso (proprio) della stringa "cgtagttcgtag" che sia anche suffisso della stringa stessa?

- R-13.3 Disegnare una figura che illustri i confronti effettuati dall'algoritmo di pattern matching a forza bruta per cercare il pattern "aabaaa" nel testo "aaabaadaabaaa".
- R-13.4 Ripetere l'esercizio precedente per l'algoritmo di Boyer-Moore, senza contare i confronti richiesti per calcolare la funzione $last(c)$.
- R-13.5 Ripetere l'Esercizio R-13.3 per l'algoritmo di Knuth-Morris-Pratt, senza contare i confronti richiesti per calcolare la funzione fallimento.
- R-13.6 Descrivere il contenuto di una mappa che rappresenti la funzione $last$ usata dall'algoritmo di pattern matching di Boyer-Moore relativamente ai caratteri del pattern seguente:

"the quick brown fox jumped over a lazy cat"

- R-13.7 Descrivere il contenuto di una tabella che rappresenti la funzione fallimento usata dall'algoritmo di Knuth-Morris-Pratt quando il pattern è la stringa "cgtaacgtcgta".
- R-13.8 Disegnare un trie standard per il seguente insieme di stringhe:

{ abab, baba, ccccc, bbaaaa, caa, bbaacc, cbcc, cbca }

- R-13.9 Disegnare un trie compresso per le stringhe usate nell'esercizio precedente.
- R-13.10 Disegnare la rappresentazione compatta del trie dei suffissi relativo alla stringa "minimize minime".
- R-13.11 Descrivere il contenuto dell'array delle frequenze e dell'albero di Huffman per la stringa seguente:

"dogs do not spot hot pots or cats"

- R-13.12 Qual è il modo migliore per moltiplicare una catena di matrici le cui dimensioni sono, in sequenza: 10×5 , 5×2 , 2×20 , 20×12 , 12×4 e 4×60 ? Dare una dimostrazione della risposta fornita.
- R-13.13 Nella Figura 13.14 abbiamo fatto vedere che GTTTAA è una sotto-seguenza di lunghezza massima comune alle stringhe X e Y là definite, anche se non si tratta dell'unica stringa possibile. Determinare un'altra sotto-seguenza comune avente la stessa lunghezza.
- R-13.14 Determinare il contenuto dell'array L che risolve, con il metodo della programmazione dinamica, il problema LCS relativo alle due stringhe:

$X = \text{"skullandbones"}$
 $Y = \text{"lullabybabies"}$

Qual è una sotto-seguenza comune a queste due stringhe che abbia lunghezza massima?

Creatività

- C-13.15 Fornire un esempio di un testo T di lunghezza n e di un pattern P di lunghezza m tale che l'algoritmo di pattern matching a forza bruta abbia un tempo d'esecuzione $\Omega(nm)$.

- C-13.16 Adattare l'algoritmo di pattern matching a forza bruta per implementare un metodo, `findLastBrute(T, P)`, che restituisca l'indice in cui si ha l'occorrenza *più a destra* del pattern P nel testo T , se ce n'è almeno una.
- C-13.17 Risolvere nuovamente il problema posto nell'esercizio precedente, adattando però l'algoritmo di pattern matching di Boyer-Moore, implementando il metodo `findLastBoyerMoore(T, P)`.
- C-13.18 Risolvere nuovamente il problema posto nell'Esercizio C-13.16, adattando però l'algoritmo di pattern matching di Knuth-Morris-Pratt, implementando il metodo `findLastKMP(T, P)`.
- C-13.19 Spiegare perché il metodo `computeFailKMP` (riportato nel Codice 13.4) viene eseguito in un tempo $O(m)$ con un pattern di lunghezza m .
- C-13.20 Dato un pattern P di lunghezza m e un testo T di lunghezza n , descrivere un metodo che in un tempo $O(n + m)$ trovi il più lungo prefisso di P che sia una sottostringa di T .
- C-13.21 Dato un pattern P di lunghezza m e un testo T di lunghezza $n > m$, diciamo che P è una sottostringa *circolare* di T se P è una sottostringa (normale) di T oppure se P è uguale alla concatenazione di un suffisso di T e di un prefisso di T , cioè se esiste un indice k , con $0 \leq k < m$, tale che $P = T[n - m + k..n - 1] + T[0..k - 1]$. Descrivere un algoritmo che in un tempo $O(n + m)$ determini se P è una sottostringa circolare di T .
- C-13.22 L'algoritmo di pattern matching di Knuth-Morris-Pratt può essere modificato per essere più veloce quando opera con stringhe binarie, ridefinendo la funzione fallimento in questo modo:

$$f(k) = \text{il massimo valore di } j < k \text{ tale che } P[0..j - 1] \hat{p}_j \text{ è un suffisso di } P[1..k],$$

dove \hat{p}_j indica il complemento del j -esimo bit di P . Descrivere come vada modificato l'algoritmo KMP perché possa trarre vantaggio da questa nuova funzione fallimento e, poi, descrivere un metodo che la calcoli. Dimostrare che tale algoritmo KMP modificato effettua al massimo n confronti tra caratteri del testo e del pattern (in confronto ai $2n$ confronti richiesti dall'algoritmo KMP standard descritto nel Paragrafo 13.2.3).

- C-13.23 Modificare l'algoritmo di Boyer-Moore semplificato che è stato presentato in questo capitolo usando un'idea simile a quella sfruttata nell'algoritmo KMP, in modo che venga eseguito in un tempo $O(n + m)$.
- C-13.24 Data una stringa di testo T di lunghezza n , descrivere un metodo che in un tempo $O(n)$ trovi il più lungo prefisso di T che sia una sottostringa della stringa inversa di T .
- C-13.25 Descrivere un algoritmo efficiente per trovare il più lungo palindromo che sia suffisso di una stringa T di lunghezza n , ricordando che un *palindromo* è una stringa che è uguale alla propria stringa inversa. Qual è il tempo d'esecuzione dell'algoritmo progettato?
- C-13.26 Descrivere un algoritmo efficiente per eliminare una stringa da un trie standard e analizzare il suo tempo d'esecuzione.
- C-13.27 Descrivere un algoritmo efficiente per eliminare una stringa da un trie compresso e analizzare il suo tempo d'esecuzione.

- C-13.28 Descrivere un algoritmo per costruire la rappresentazione compatta di un trie di suffissi, data la sua rappresentazione non compatta, e analizzare il suo tempo d'esecuzione.
- C-13.29 Progettare una classe che implementi un trie standard per un insieme di stringhe. La classe deve avere un costruttore che riceve come argomento un elenco di stringhe e un metodo che verifichi se una determinata stringa sia memorizzata all'interno del trie.
- C-13.30 Progettare una classe che implementi un trie compresso per un insieme di stringhe. La classe deve avere un costruttore che riceve come argomento un elenco di stringhe e un metodo che verifichi se una determinata stringa sia memorizzata all'interno del trie.
- C-13.31 Progettare una classe che implementi un trie di prefissi per un stringa data. La classe deve avere un costruttore che riceve come argomento una stringa e un metodo che effettui il pattern matching sulla stringa.
- C-13.32 Data una stringa X di lunghezza n e una stringa Y di lunghezza m , descrivere un algoritmo che in un tempo $O(n + m)$ trovi il più lungo prefisso di X che sia anche un suffisso di Y .
- C-13.33 Descrivere un algoritmo greedy efficiente che, data una somma da pagare e una somma pagata, generi il resto usando il numero minimo di monete, nell'ipotesi che ce ne siano di quattro valori, in centesimi: 25, 10, 5 e 1 (i cui nomi sono, rispettivamente, *quarter*, *dime*, *nickel* e *penny*). Spiegare perché l'algoritmo sviluppato è corretto.
- C-13.34 Descrivere un esempio di insieme di valori di monete usando il quale un algoritmo greedy non è in grado di risolvere il problema precedente usando sempre il minimo numero di monete.
- C-13.35 Nel problema della sorveglianza di una galleria d'arte, ci troviamo di fronte a un segmento L che rappresenta un lungo corridoio all'interno di una galleria d'arte. Dato un insieme di numeri reali $X = \{x_0, x_1, \dots, x_{n-1}\}$ che rappresentano le posizioni dei dipinti lungo il corridoio, supponiamo che un sorvegliante sia in grado di proteggere tutti i dipinti che si trovano a una distanza non maggiore di un'unità dalla sua posizione (in entrambe le direzioni). Progettare un algoritmo che trovi un posizionamento dei sorveglianti che protegga tutti i dipinti presenti in X impiegando il minimo numero di sorveglianti.
- C-13.36 Anna ha appena vinto una gara e, come premio, può prendere gratis n dolcetti, scegliendoli in un negozio. Anna non è così giovane da non sapere che alcuni dolcetti sono più costosi di altri. I barattoli in cui sono conservati i dolcetti sono numerati: $0, 1, \dots, m - 1$, in modo che il barattolo j contiene n_j dolcetti, con prezzo unitario c_j . Progettare un algoritmo che in un tempo $O(n + m)$ consenta ad Anna di rendere massimo il valore dei dolcetti che sceglie come premio, dimostrandone la correttezza.
- C-13.37 Implementare una schema di compressione e decompressione del testo che sia basato sulla codifica di Huffman.
- C-13.38 Progettare un algoritmo efficiente per il problema della moltiplicazione matriciale a catena che visualizzi un'espressione, completa di parentesi, che indichi come vadano moltiplicate le matrici della catena per effettuare il numero minimo di moltiplicazioni scalari.

- C-13.39 Un nativo australiano di nome Anatjari vuole attraversare un deserto portando con sé una sola bottiglia d'acqua, disponendo di una mappa che segnala tutte le fonti d'acqua presenti lungo il percorso. Nell'ipotesi che possa camminare per k miglia usando una sola bottiglia d'acqua, progettare un algoritmo efficiente per determinare dove Anatjari possa riempire la sua bottiglia in modo da fare il minimo numero di soste. Dimostrare la correttezza dell'algoritmo proposto.
- C-13.40 Data una sequenza $S = (x_0, x_1, \dots, x_{n-1})$ di numeri, descrivere un algoritmo che in un tempo $O(n^2)$ trovi la più lunga sotto-sequenza $T = (x_{i_0}, x_{i_1}, \dots, x_{i_{k-1}})$ di numeri appartenenti a S tale che $i_j < i_{j+1}$ e che $x_{i_j} > x_{i_{j+1}}$. In pratica, T deve essere la più lunga sotto-sequenza di S contenente numeri in ordine decrescente.
- C-13.41 Dato un poligono convesso P , una *triangolazione* di P è l'aggiunta di diagonali che collegano vertici di P in modo che ogni figura geometrica interna sia un triangolo. Il *peso* (weight) di una triangolazione è la somma delle lunghezze delle diagonali che la compongono. Nell'ipotesi che si possa calcolare qualunque lunghezza e che le si possano sommare e confrontare in un tempo costante, descrivere un algoritmo efficiente per calcolare una triangolazione di P di peso minimo.
- C-13.42 Descrivere un algoritmo efficiente per determinare se un pattern P è una sotto-sequenza (non solo una sottostringa) di un testo T . Qual è il tempo d'esecuzione dell'algoritmo proposto?
- C-13.43 Un'azione di modifica di un testo (un *edit*) è costituita dall'inserimento, dalla rimozione o dalla sostituzione di un singolo carattere. Definiamo *distanza di modifica* (*edit distance*) tra le due stringhe X e Y di lunghezza, rispettivamente, n e m , il minimo numero di modifiche che trasformano X in Y . Ad esempio, le stringhe "algorithm" e "rhythm" hanno distanza di modifica uguale a 6. Progettare un algoritmo che in un tempo $O(nm)$ calcoli la distanza di modifica tra X e Y .
- C-13.44 Scrivere un programma che, date due stringhe di caratteri (che, ad esempio, potrebbero rappresentare porzioni di DNA), calcoli la loro distanza di modifica, basandosi sull'algoritmo progettato nell'esercizio precedente.
- C-13.45 Date le stringhe X e Y di lunghezza, rispettivamente, n e m , indichiamo con $B(j, k)$ la lunghezza della più lunga sottostringa comune ai suffissi $X[n - j..n - 1]$ e $Y[m - k..m - 1]$. Progettare un algoritmo che in un tempo $O(nm)$ calcoli tutti i valori di $B(j, k)$, con $j = 1, \dots, n$ e $k = 1, \dots, m$.
- C-13.46 Dati tre array di numeri interi, A , B e C , ciascuno di lunghezza n , e dato un numero intero k , progettare un algoritmo che in un tempo $O(n^2 \log n)$ determini se esistono tre numeri a in A , b in B e c in C , tali che $k = a + b + c$.
- C-13.47 Descrivere un algoritmo che risolva il problema precedente in un tempo $O(n^2)$.

Progettazione

- P-13.48 Sulla base del numero di confronti eseguiti, effettuare un'analisi sperimentale dell'efficienza degli algoritmi di pattern matching a forza bruta e KMP, al variare della lunghezza del pattern.
- P-13.49 Sulla base del numero di confronti eseguiti, effettuare un'analisi sperimentale dell'efficienza degli algoritmi di pattern matching a forza bruta e di Boyer-Moore, al variare della lunghezza del pattern.
- P-13.50 Effettuare un confronto sperimentale delle velocità relative degli algoritmi di pattern matching a forza bruta, KMP e di Boyer-Moore. Riportare i tempi d'esecuzione.

cuzione relativi quando tali algoritmi agiscono su documenti di grandi dimensioni, al variare della lunghezza del pattern.

- P-13.51 Condurre esperimenti relativi all'efficienza del metodo `indexOf` della classe `String` della libreria di Java e sviluppare un'ipotesi in merito all'algoritmo di pattern matching utilizzato al suo interno. Descrivere gli esperimenti condotti e le conclusioni tratte.
- P-13.52 Un algoritmo di pattern matching molto efficiente, sviluppato da Rabin e Karp [54], si basa sull'uso della tecnica di hashing e ottiene prestazioni attese molto buone. Ricordiamo che l'algoritmo a forza bruta confronta il pattern con ogni suo possibile posizionamento nel testo, impiegando un tempo $O(m)$, nel caso peggiore, per ognuno di tali confronti. Il presupposto dell'algoritmo di Rabin-Karp è quello di calcolare una funzione di hash, $h(\cdot)$, del pattern di lunghezza m , per poi calcolare la funzione di hash per tutte le sottostringhe del testo aventi lunghezza m . Il pattern P è uguale alla sottostringa $T[j..i+m-1]$ solo se $h(P) = h(T[j..i+m-1])$. Se i valori della funzione di hash sono uguali, l'effettiva corrispondenza della sottostringa con il pattern deve essere verificata con l'approccio a forza bruta, perché esiste la possibilità che avvenga una collisione accidentale dei valori di hash di stringhe diverse, ma, con una buona funzione di hash, queste "false corrispondenze" saranno pochissime.

Il problema successivo, tuttavia, è che il calcolo di una buona funzione di hash per una sottostringa di lunghezza m richiederebbe probabilmente un tempo $O(m)$. Se facessimo il calcolo per ciascuno dei possibili posizionamenti del pattern, il cui numero è $O(n)$, l'algoritmo non sarebbe migliore di quello a forza bruta. Il trucco sta nell'utilizzo di un *codice di hash polinomiale*, definito nel Paragrafo 10.2.1 in questo modo:

$$(x_0a^{m-1} + x_1a^{m-2} + \dots + x_{m-2}a + x_{m-1}) \bmod p$$

per una sottostringa $(x_0, x_1, \dots, x_{m-1})$, dove a è un numero casuale e p è un numero primo di valore elevato. Usando la formula seguente, si può calcolare in un tempo $O(1)$ il valore di hash di ciascuna successiva sottostringa del testo:

$$h(T[j+1..j+m]) = (a \cdot h(T[j..j+m-1]) - x_j a^m + x_{j+m}) \bmod p$$

Implementare l'algoritmo di Rabin-Karp e valutarne l'efficienza.

- P-13.53 Implementare il semplice motore di ricerca descritto nel Paragrafo 13.3.4 per le pagine di un piccolo sito web. Usare tutte le parole presenti nelle pagine del sito come termini indice, escludendo le *stop word* come articoli, preposizioni e pronomi.
- P-13.54 Implementare un motore di ricerca per le pagine di un piccolo sito web aggiungendo una funzione di *ranking* al motore semplificato descritto nel Paragrafo 13.3.4. La funzione di *page-ranking*, che attribuisce un punteggio alle singole pagine, deve fare in modo che le pagine più rilevanti per una determinata ricerca vengano restituire per prime. Usare tutte le parole presenti nelle pagine del sito come termini indice, escludendo le *stop word* come articoli, preposizioni e pronomi.
- P-13.55 Usare l'algoritmo LCS per calcolare il miglior allineamento tra alcune stringhe di DNA, ottenibili consultando in Internet la base di dati GenBank.

P-13.56 Sviluppare un correttore ortografico (*spell checker*) che usi la distanza di modifica (*edit distance*) definita nell'Esercizio C-13.43 per determinare quali parole corrette si avvicino maggiormente a una parola non corretta, in modo da poter fornire suggerimenti di correzione.

Note

L'algoritmo KMP fu descritto da Knuth, Morris e Pratt nel loro articolo su rivista [62], mentre Boyer e Moore descrissero il loro algoritmo in un altro articolo pubblicato nello stesso anno [17]. Nel loro articolo, però, Knuth e colleghi [62] dimostrarono anche che l'algoritmo di Boyer-Moore era caratterizzato da un tempo d'esecuzione lineare. Più recentemente, Cole [23] ha dimostrato che l'algoritmo di Boyer-Moore effettua, nel caso peggiore, al massimo $3n$ confronti tra caratteri, e che questo limite superiore è stretto. Tutti gli algoritmi presentati in questo capitolo sono anche discussi nel libro di Aho [4], anche se in un modo più teorico, includendo i metodi per il pattern matching di espressioni canoniche (*regular expression*). Al lettore che fosse interessato all'approfondimento degli studi nel campo degli algoritmi di pattern matching suggeriamo il libro di Stephen [84] e alcuni capitoli del libro di Aho [4] e di Crochemore e Lecroq [26]. Il trie è stato inventato da Morrison [74] e discusso approfonditamente nel classico testo di Knuth, *Sorting and Searching* [61]. Il nome "Patricia" è un acronimo di "Practical Algorithm To Retrieve Information Coded In Alphanumeric" [74]. McCreight [68] ha dimostrato come costruire un trie dei suffissi in un tempo lineare. La programmazione dinamica fu sviluppata nella comunità scientifica della ricerca operativa e formalizzata da Bellman [12].

14

Algoritmi per grafi

14.1 Grafi

Un **grafo** (*graph*) è un modo per rappresentare relazioni esistenti tra coppie di oggetti. Un **grafo**, cioè, è un insieme di oggetti, chiamati **vertici** (*vertex*), e una raccolta di accoppiamenti tra loro, chiamati **lati** (*edge*). I grafi hanno applicazioni in molti ambiti, tra i quali possiamo citare i trasporti pubblici, le reti di calcolatori, le reti di distribuzione dell'energia elettrica e le mappe di navigazione stradale. Forse è meglio sottolineare fin da subito che il "grafo" non va confuso in alcun modo con i diagrammi a barre o i grafici di funzioni, che sono, appunto, "grafici", non "grafo", e non hanno niente a che vedere con l'argomento trattato in questo capitolo.

Considerato in astratto, un **grafo** G è semplicemente un insieme V di **vertici** e una raccolta E di coppie di vertici appartenenti a V , dette **lati**. Quindi, un grafo è un modo per rappresentare connessioni o relazioni tra coppie di oggetti appartenenti a un insieme V . Incidentalmente, osserviamo che alcuni testi usano una diversa terminologia per i grafi e chiamano **nodi** e **archi** quelli che noi chiamiamo, rispettivamente, vertici e lati. In questo libro useremo i termini "vertici" e "lati".

I lati di un grafo possono essere **orientati** (*directed*) oppure **non orientati** (*undirected*). Un lato (u, v) è detto **orientato** da u a v se la coppia (u, v) è una coppia orientata, con u che precede v . Un lato (u, v) si dice, invece, **non orientato** se la coppia (u, v) è una coppia non orientata. A volte per i lati non orientati si usa la notazione di insieme, $\{u, v\}$, ma per semplicità useremo comunque la notazione di coppia (u, v) , ricordando che, nel caso in cui la coppia non sia orientata, (u, v) è equivalente a (v, u) . Solitamente i grafi vengono visualizzati disegnandone i vertici sotto forma di ovali o rettangoli, mentre i lati sono rappresentati da segmenti o archi che collegano coppie di vertici, cioè di ovali o rettangoli. Ecco alcuni esempi di grafi, orientati e non orientati.

Esempio 14.1: Le collaborazioni tra i ricercatori di una certa disciplina si possono visualizzare costruendo un grafo, i cui vertici siano associati ai ricercatori stessi e i cui lati colleghino coppie di vertici associati a ricercatori che hanno scritto insieme almeno un lavoro su rivista o un libro, cioè sono coautori (si veda la Figura 14.1). Tali lati sono non orientati, perché la relazione “essere coautore di” è una relazione simmetrica: se A ha scritto qualcosa insieme con B , allora necessariamente B ha scritto qualcosa insieme con A .

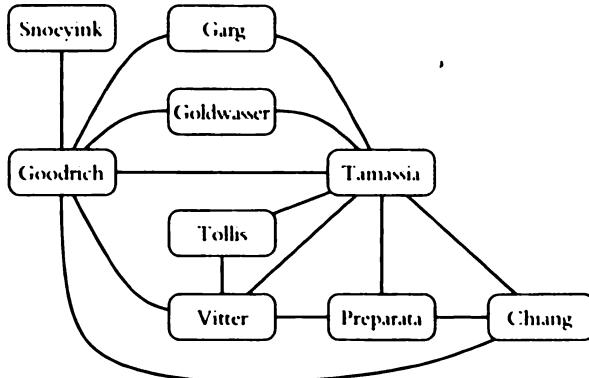


Figura 14.1: Grafo che rappresenta, tra alcuni autori di lavori scientifici, la relazione “essere coautore di”.

Esempio 14.2: A un programma orientato agli oggetti può essere associato un grafo i cui vertici rappresentano le classi definite nel programma e i cui lati indicano le relazioni di ereditarietà tra le classi. Esiste un lato che va dal vertice v al vertice u se e solo se la classe v deriva dalla classe u . Tali lati sono chiaramente orientati, perché la relazione di ereditarietà va in una sola direzione (cioè è asimmetrica).

Se tutti i lati di un grafo sono non orientati, allora diciamo che il grafo stesso è un **grafo non orientato** (*undirected graph*). Analogamente, un grafo i cui lati siano tutti orientati è detto **grafo orientato** (*directed graph*, o *digraph*, *digraph*). Un grafo che ha sia lati orientati sia lati non orientati viene spesso chiamato **grafo misto** (*mixed graph*). Osserviamo che un grafo non orientato o un grafo misto può sempre essere convertito in un grafo orientato, sostituendo ogni lato non orientato (u, v) con la coppia di lati orientati (u, v) e (v, u) . Spesso è utile, però, lasciare i grafi non orientati o misti così come sono, perché anche tali grafi sono utili in molte applicazioni, come nell'esempio che segue.

Esempio 14.3: Una mappa stradale può essere rappresentata da un grafo i cui vertici sono gli incroci (o i vicoli ciechi) e i cui lati sono tratti di strada privi di incroci. Questo grafo ha sia lati non orientati, che rappresentano strade a doppio senso di marcia, sia lati orientati, che corrispondono a strade a senso unico. Quindi, una mappa stradale rappresentata in questo modo è un grafo misto.

Esempio 14.4: Esempi concreti di grafi sono presenti negli edifici sotto forma di reti elettriche o reti di tubature. Tali reti possono essere viste come grafi, dove ogni connettore, raccordo, presa elettrica o

rubinetto idraulico è un vertice e ogni tratto non interrotto di filo o di tubo è un lato. Tali grafi fanno, poi, parte di grafi di maggiori dimensioni, che rappresentano la rete di distribuzione elettrica e idraulica del quartiere. In relazione ai particolari aspetti della rete a cui siamo interessati, potremo considerare i lati di questi grafi come orientati o non orientati, perché, almeno in linea di principio, l'acqua può scorrere in un tubo e la corrente elettrica può fluire in un filo in entrambe le direzioni.

Due vertici collegati da un lato sono i suoi *vertici terminali* (o *punti terminali*, *endpoint*). Se un lato è orientato, il suo primo vertice è la sua *origine*, mentre l'altro è la sua *destinazione*. Due vertici u e v si dicono *adiacenti* (*adjacent*) se esiste un lato i cui vertici terminali siano u e v . Un lato si dice *incidente* in un vertice se questo è uno dei suoi due vertici terminali. I *lati uscenti* (*outgoing edge*) da un vertice sono i lati orientati la cui origine è quel vertice. I *lati entranti* (*ingoing edge*) in un vertice sono i lati orientati la cui destinazione è quel vertice. Il *grado* (*degree*) di un vertice v , indicato con $\deg(v)$, è il numero di lati incidenti in v . Il *grado entrante* (*in-degree*) e il *grado uscente* (*out-degree*) di un vertice v sono il numero di lati entranti in v e, rispettivamente, uscenti da v ; si indicano, rispettivamente, con $\text{indeg}(v)$ e $\text{outdeg}(v)$.

Esempio 14.5: Possiamo studiare il trasporto aereo costruendo un grafo G , detto *flight network* ("rete di voli"), i cui vertici sono associati agli aeroporti e i cui lati rappresentano voli (come si può vedere nella Figura 14.2). I lati del grafo G sono orientati, perché ciascun volo ha una specifica direzione di viaggio. I punti terminali di un lato e di G corrispondono, rispettivamente, all'aeroporto di partenza e di destinazione del volo corrispondente a e . Due aeroporti sono adiacenti in G se esiste un volo tra di loro e un lato e è incidente in un vertice v in G se il volo e parte o arriva dall'aeroporto associato a v . I lati uscenti da un vertice v corrispondono ai voli in partenza dall'aeroporto v , mentre i lati entranti in v corrispondono ai voli in arrivo all'aeroporto v . Infine, il grado entrante di un vertice v di G è uguale al numero di voli in arrivo all'aeroporto v , così come il grado uscente di v è uguale al numero di voli in partenza da v .

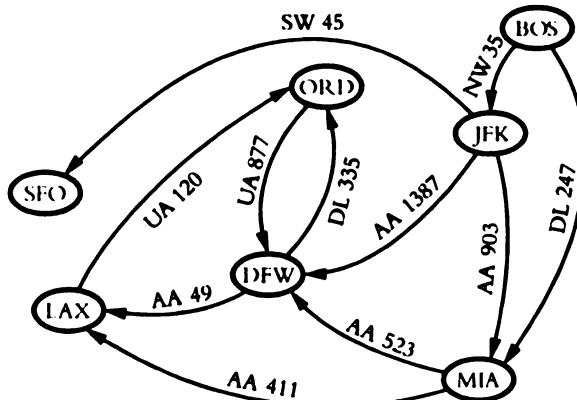


Figura 14.2: Esempio di grafo orientato che rappresenta una *flight network*, cioè una rete di voli. I vertici terminali del lato UA 120 sono LAX e ORD, cioè gli aeroporti di Los Angeles e Chicago, che sono, quindi, adiacenti. Il grado entrante di DFW (l'aeroporto di Dallas) è uguale a 3, mentre il suo grado uscente è 2.

La definizione di grafo fa riferimento ai gruppi di lati come a *collezioni* o raccolte (*collection*), non a *insiemi*, per cui è ammesso che due lati non orientati abbiano gli stessi vertici terminali, così come due lati orientati possono avere la stessa origine e la stessa destinazione. Lati di questo tipo si dicono *lati paralleli* o *lati multipli*. Una *flight network* (definita nell'Esempio 14.5) può contenere lati paralleli: lati multipli tra la stessa coppia di vertici possono rappresentare voli diversi che operano sulla stessa rotta in momenti diversi della giornata. Un altro lato di tipo speciale è quello che collega un vertice a se stesso: un lato (orientato o non orientato) i cui due vertici terminali coincidono si chiama *auto-anello* (*self-loop*) del grafo. Se pensiamo al grafo associato a una mappa stradale (così come definito nell'Esempio 14.3), un auto-anello può corrispondere a una strada curva che torna al suo punto di partenza, un elemento stradale che consente di fare un'inversione di marcia in modo sicuro.

Con poche eccezioni, i grafi solitamente non hanno lati paralleli o auto-anelli: grafi con queste caratteristiche si dicono *semplici*. Quindi, possiamo dire che i lati di un grafo semplice sono un *insieme* di coppie di vertici (e non solo una generica collezione). In tutto questo capitolo, salvo dove diversamente specificato, ipotizzeremo che i grafi siano semplici.

Un *percorso* (*path*) è una sequenza di vertici e lati alternati che inizia con un vertice e termina con un vertice, tale che ogni lato sia incidente nel vertice che lo precede e nel vertice che lo segue lungo il percorso. Un *ciclo* (*cycle*) è un percorso che inizia e finisce nello stesso vertice e che contiene almeno un lato. Diciamo che un percorso è *semplice* se tutti i suoi vertici sono distinti, mentre diciamo che un ciclo è *semplice* se tutti i suoi vertici sono distinti, tranne il primo e l'ultimo, che coincidono. Un *percorso orientato* (*directed path*) è un percorso in cui tutti i lati sono orientati e vengono attraversati lungo la propria direzione; un *ciclo orientato* (*directed cycle*) è definito in modo analogo. Ad esempio, con riferimento alla Figura 14.2, (BOS, NW 35, JFK, AA 1387, DFW) è un percorso semplice e orientato, mentre (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) è un ciclo semplice e orientato. Osserviamo che un grafo orientato può avere un ciclo costituito da due lati aventi direzione opposta tra la stessa coppia di vertici, come, ad esempio, il ciclo (ORD, UA 877, DFW, DL 335, ORD) nella Figura 14.2. Un grafo orientato è *aciclico* (*acyclic*) se non contiene cicli orientati. Ad esempio, se eliminassimo il lato UA 877 dal grafo della Figura 14.2, il grafo rimanente sarebbe aciclico. Se un grafo è semplice, quando descriviamo un percorso *P* o un ciclo *C* possiamo evitare di scrivere i suoi lati, perché sono definiti univocamente.

Esempio 14.6: Dato un grafo *G* che rappresenti una mappa stradale (come descritto nell'Esempio 14.3), possiamo definire un modello di una coppia di persone che stanno guidando per andare a cena nel loro ristorante preferito mediante un percorso in *G*. Se conoscono la strada, non passeranno due volte per lo stesso incrocio, quindi seguiranno un percorso semplice in *G*. Analogamente, l'intero viaggio della coppia, da casa al ristorante e ritorno, può essere rappresentato da un ciclo. Se fanno il viaggio di andata lungo una strada completamente diversa da quella del viaggio di ritorno, allora l'intero viaggio sarà un ciclo semplice. Infine, se percorrono soltanto strade a senso unico, il loro viaggio sarà un ciclo orientato.

Dati i vertici *u* e *v* in un grafo (orientato) *G*, diciamo che *u* raggiunge *v*, e che *v* è raggiungibile da *u*, se *G* contiene un lato (orientato) che va da *u* a *v*. In un grafo non orientato, il concetto di raggiungibilità è simmetrico, cioè *u* raggiunge *v* se e solo se *v* può raggiungere *u*. Tuttavia, in un grafo orientato, può capitare che *u* raggiunga *v* senza che *v* raggiunga *u*, perché un percorso orientato deve essere attraversato seguendo le direzioni dei suoi lati.

Un grafo è **connesso** (*connected*) se esiste un percorso tra qualsiasi coppia dei suoi vertici. Un grafo orientato G è **fortemente connesso** (*strongly connected*) se, per ogni coppia di suoi vertici u e v , si ha che u raggiunge v e v raggiunge u . La Figura 14.3 riporta alcuni esempi di queste situazioni.

Un **sotto-grafo** (*subgraph*) di un grafo G è un grafo H i cui insiemi di vertici e lati sono sottoinsiemi, rispettivamente, degli insiemi di vertici e lati di G . Un **sotto-grafo ricoprente** (*spanning subgraph*) di G è un sotto-grafo di G che contiene tutti i vertici del grafo G . Se un grafo G non è connesso, i suoi sotto-grafi connessi massimali (cioè di dimensioni massime) sono detti **componenti connessi** di G . Una **foresta** (*forest*) è un grafo privo di cicli. Un **albero** (*tree*) è una foresta connessa, cioè è un grafo connesso privo di cicli (osserviamo che questa definizione di albero è in qualche modo diversa da quella che abbiamo dato nel Capitolo 8, perché in questo caso non è necessario designare una radice). Un **albero ricoprente** (*spanning tree*) di un grafo è un sotto-grafo ricoprente che sia anche un albero.

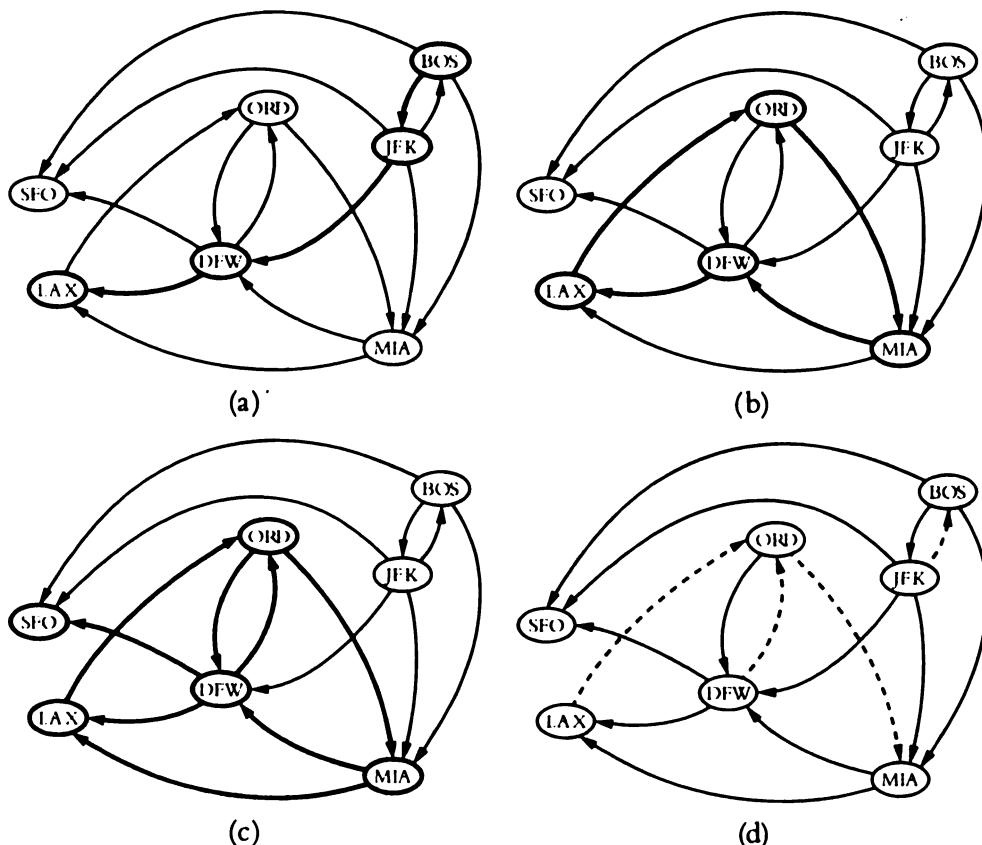


Figura 14.3: Esempi di raggiungibilità in un grafo orientato: (a) è evidenziato un percorso orientato da BOS a LAX; (b) è evidenziato il ciclo orientato (ORD, MIA, DFW, LAX, ORD) e i suoi vertici definiscono un sotto-grafo fortemente connesso; (c) è evidenziato il sotto-grafo contenente tutti i vertici raggiungibili da ORD e i lati che li connettono; (d) la rimozione dei lati tratteggiati produce un grafo orientato aciclico.

Esempio 14.7: Forse il grafo di cui oggi si parla di più è Internet, che può essere vista come un grafo i cui vertici sono computer e i cui lati (non orientati) sono connessioni di comunicazione tra coppie di computer di Internet. I computer e le connessioni che li coinvolgono all'interno di un unico dominio, come wiley.com, formano un sotto-grafo di Internet. Se questo sotto-grafo è connesso, allora due utenti di computer che si trovino in tale dominio possono scambiarsi messaggi di posta elettronica senza che i loro pacchetti di informazioni debbano uscire dal loro dominio. Supponiamo che i lati di questo sotto-grafo formino un albero ricoprente: questo implica che, se anche una sola connessione si interrompe (ad esempio perché qualcuno stacca il cavo di comunicazione dal retro di un computer del dominio), allora questo sotto-grafo non sarà più connesso.

Nelle proposizioni che seguono, analizziamo alcune proprietà importanti dei grafi.

Proposizione 14.8: Se G è un grafo con m lati e con un insieme di vertici V , allora

$$\sum_{v \text{ in } V} \deg(v) = 2m.$$

Dimostrazione: Un lato (u, v) è contato due volte nella sommatoria: una volta per il suo vertice terminale u e una volta per l'altro vertice terminale v . Quindi, il contributo complessivo dei lati ai gradi dei vertici è il doppio del numero di lati. ■

Proposizione 14.9: Se G è un grafo orientato con m lati e con un insieme di vertici V , allora

$$\sum_{v \text{ in } V} \text{indeg}(v) = \sum_{v \text{ in } V} \text{outdeg}(v) = m.$$

Dimostrazione: In un grafo orientato, un lato (u, v) contribuisce per un'unità al grado uscente del suo vertice iniziale u e per un'unità al grado entrante del suo vertice finale v . Quindi, il contributo complessivo dei lati ai gradi uscenti dei vertici è uguale al numero di lati, così come lo è il contributo ai gradi entranti. ■

Ora dimostreremo che un grafo semplice con n vertici ha un numero di lati $O(n^2)$.

Proposizione 14.10: Sia G un grafo semplice con n vertici e m lati. Se G è non orientato, allora $m \leq n(n - 1)/2$, mentre se è orientato $m \leq n(n - 1)$.

Dimostrazione: Supponiamo prima che G sia non orientato. Dato che non possono esistere due lati che abbiano la stessa coppia di vertici terminali e dato che non sono presenti auto-anelli, in questo caso il massimo grado di un vertice in G è $n - 1$. Quindi, per la Proposizione 14.8, si ha che $2m \leq n(n - 1)$, cioè $m \leq n(n - 1)/2$. Supponiamo, ora, che G sia orientato. Dato che non possono esistere due lati che abbiano la stessa origine e la stessa destinazione e dato che non sono presenti auto-anelli, in questo caso il massimo grado entrante di un vertice in G è $n - 1$. Quindi, per la Proposizione 14.9, si ha che $m \leq n(n - 1)$. ■

Ci sono, poi, molte proprietà interessanti che riguardano gli alberi, le foreste e i grafî connessi.

Proposizione 14.11: *Sia G un grafo non orientato, con n vertici e m lati.*

- *Se G è connesso, allora $m \geq n - 1$.*
- *Se G è un albero, allora $m = n - 1$.*
- *Se G è una foresta, allora $m \leq n - 1$.*

14.1.1 Il grafo come tipo di dato astratto

Un grafo è una collezione di vertici e lati. Come modello di questa astrazione usiamo una combinazione di tre tipi di dati: **Vertex**, **Edge** e **Graph**. Un oggetto di tipo **Vertex**, che rappresenta un vertice, è un oggetto molto semplice, che memorizza un elemento arbitrario fornito dall'utilizzatore (ad esempio, un codice aeroportuale), che ipotizziamo possa essere ispezionato con il metodo `getElement()`. Anche un oggetto di tipo **Edge**, che rappresenta un lato, può memorizzare un oggetto associato al lato (ad esempio, un numero di volo, una distanza di viaggio o un costo), restituito dal suo metodo `getElement()`.

L'astrazione principale per rappresentare i grafî è, quindi, il tipo di dato astratto **Graph**. Assumiamo che un grafo possa essere *non orientato* oppure *orientato*, decisione che viene presa all'atto della costruzione di un esemplare, ricordando che un grafo misto può essere rappresentato con un grafo orientato, trasformando ciascun lato $\{u, v\}$ non orientato in una coppia di lati orientati, (u, v) e (v, u) . Il tipo di dato astratto “grafo” contiene i metodi seguenti:

- numVertices():** Restituisce il numero di vertici del grafo.
- vertices():** Restituisce un contenitore iterabile che consente di scandire tutti i vertici del grafo.
- numEdges():** Restituisce il numero di lati del grafo.
- edges():** Restituisce un contenitore iterabile che consente di scandire tutti i lati del grafo.
- getEdge(u, v):** Restituisce il lato che va dal vertice u al vertice v , se esiste, altrimenti `null`. In un grafo non orientato, non c'è alcuna differenza tra `getEdge(u, v)` e `getEdge(v, u)`.
- endVertices(e):** Restituisce un array contenente i due vertici terminali del lato e . Se il grafo è orientato, il primo vertice dell'array è l'origine del lato e il secondo è la sua destinazione.
- opposite(v, e):** Restituisce il vertice terminale di e diverso da v ; si verifica un errore se e non è incidente in v .
- outDegree(v):** Restituisce il numero di lati uscenti dal vertice v .
- inDegree(v):** Restituisce il numero di lati entranti nel vertice v . In un grafo non orientato, questo metodo restituisce lo stesso valore che viene restituito da `outDegree(v)`.
- outgoingEdges(v):** Restituisce un contenitore iterabile che consente di scandire tutti i lati uscenti dal vertice v .

incomingEdges(v): Restituisce un contenitore iterabile che consente di scandire tutti i lati entranti nel vertice v . In un grafo non orientato, questo metodo restituisce lo stesso contenitore che viene restituito da **outgoingEdges(v)**.

insertVertex(x): Crea e restituisce un nuovo oggetto di tipo **Vertex** che memorizza l'elemento x .

insertEdge(u, v, x): Crea e restituisce un nuovo oggetto di tipo **Edge** che memorizza l'elemento x e rappresenta un lato che va dal vertice u al vertice v ; si verifica un errore se esiste già un lato che va da u a v .

removeVertex(v): Elimina dal grafo il vertice v e tutti i lati incidenti in esso.

removeEdge(e): Elimina dal grafo il lato e .

14.2 Strutture dati per grafi

In questo paragrafo presenteremo quattro strutture dati adatte a rappresentare grafi. In ciascuna rappresentazione gestiremo un contenitore che memorizza i vertici del grafo, ma le quattro strutture differiscono fortemente rispetto al modo in cui organizzano i lati al proprio interno.

- In una struttura a **lista di lati** (*edge list*), usiamo una lista non ordinata contenente tutti i lati. Questo è il minimo indispensabile, ma non consente di individuare in modo efficiente un lato specifico, (u, v) , né l'insieme di tutti i lati incidenti in un vertice v .
- In una struttura a **lista di adiacenze** (*adjacency list*), aggiungiamo, per ciascun vertice, un'ulteriore lista contenente i lati incidenti in quel vertice. Questo consente di trovare in modo più efficiente l'insieme di tutti i lati incidenti in un vertice dato.
- Una **mappa di adiacenze** (*adjacency map*) è simile a una lista di adiacenze, ma il contenitore secondario di tutti i lati incidenti in un vertice è organizzato come una mappa, invece che come una lista, con il vertice adiacente utilizzato come chiave. Questo consente un più efficiente accesso a un lato specifico, (u, v) , che, ad esempio, può avvenire in un tempo atteso $O(1)$ se la mappa usa una tecnica di hashing.
- Una **matrice di adiacenze** (*adjacency matrix*) consente di accedere a un lato specifico, (u, v) , in un tempo $O(1)$ nel caso peggiore, usando una matrice $n \cdot n$ per un grafo avente n vertici. Ogni cella della matrice è dedicata a memorizzare un riferimento al lato (u, v) per una specifica coppia di vertici u e v ; se tale lato non esiste, la cella conterrà **null**.

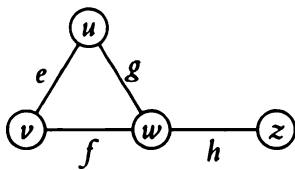
La Tabella 14.1 riassume le prestazioni di queste strutture.

14.2.1 La struttura a lista di lati (*edge list*)

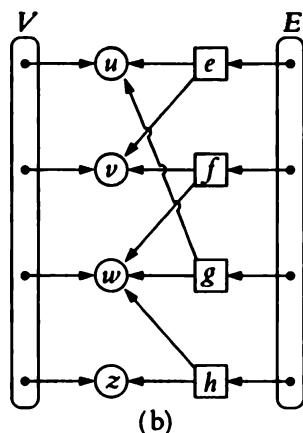
La struttura a **lista di lati** (*edge list*) è probabilmente la più semplice rappresentazione possibile per un grafo G , anche se non la più efficiente. Tutti gli oggetti di tipo "vertice" sono memorizzati in una lista non ordinata V e tutti gli oggetti di tipo "lato" sono memorizzati in un'analogia lista non ordinata E , come si può vedere, in un esempio, nella Figura 14.4.

Tabella 14.1: Riassunto dei tempi d'esecuzione dei metodi dell'ADT "grafo", ottenibili usando le rappresentazioni discusse in questo paragrafo. Abbiamo indicato con n il numero di vertici, con m il numero di lati e con d_v il grado del vertice v . Si osservi che la matrice di adiacenze occupa uno spazio $O(n^2)$, mentre per le altre strutture lo spazio è $O(n + m)$.

Metodo	Lista di dati	Lista di adiacenze	Mappa di adiacenze	Matrice di adiacenze
<code>numVertices()</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>numEdges()</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>vertices()</code>	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<code>edges()</code>	$O(m)$	$O(m)$	$O(m)$	$O(m)$
<code>getEdge(u, v)</code>	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ atteso	$O(1)$
<code>outDegree(v)</code>	$O(m)$	$O(1)$	$O(1)$	$O(n)$
<code>inDegree(v)</code>				
<code>outgoingEdges(v)</code>	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
<code>incomingEdges(v)</code>				
<code>insertVertex(x)</code>	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
<code>removeVertex(v)</code>	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
<code>insertEdge(u, v, x)</code>	$O(1)$	$O(1)$	$O(1)$ atteso	$O(1)$
<code>removeEdge(e)</code>	$O(1)$	$O(1)$	$O(1)$ atteso	$O(1)$



(a)



(b)

Figura 14.4: (a) Un grafo G ; (b) la rappresentazione schematica di G mediante la struttura a lista di lati. Osserviamo che un oggetto di tipo "lato" contiene riferimenti ai due oggetti di tipo "vertice" che corrispondono ai suoi vertici terminali, ma i vertici non contengono alcun riferimento ai lati.

Per fornire supporto ai molti metodi del tipo di dato astratto Graph (presentato nel Paragrafo 14.1), ipotizziamo che questa struttura abbia le seguenti ulteriori caratteristiche. I contenitori V e E sono rappresentati con liste doppiamente concatenate, usando esemplari della nostra classe `LinkedPositionalList`, vista nel Capitolo 7.

Oggetti di tipo "vertice"

L'oggetto che rappresenta un vertice v contenente l'elemento x dispone delle seguenti variabili di esemplare:

- Un riferimento all'elemento x , per consentire l'esecuzione del metodo `getElement()`.
- Un riferimento alla posizione del vertice nella lista V , consentendo così che v possa essere eliminato facilmente da V nel momento in cui debba essere eliminato dal grafo.

Oggetti di tipo "lato"

L'oggetto che rappresenta un lato e contenente l'elemento x dispone delle seguenti variabili di esemplare:

- Un riferimento all'elemento x , per consentire l'esecuzione del metodo `getElement()`.
- Riferimenti agli oggetti di tipo "vertice" associati ai vertici terminali di e . Questo consente di realizzare i metodi `endVertices(e)` e `opposite(v, e)` in modo che vengano eseguiti in un tempo costante.
- Un riferimento alla posizione del lato nella lista E , consentendo così che e possa essere eliminato facilmente da E nel momento in cui debba essere eliminato dal grafo.

Prestazioni della struttura a lista di lati

Le prestazioni della struttura a lista di lati per la realizzazione del tipo di dato astratto "grafo" sono riassunte nella Tabella 14.2. Iniziamo la discussione dall'occupazione di spazio in memoria, che è $O(n + m)$ per la rappresentazione di un grafo avente n vertici e m lati. Ogni singolo esemplare di vertice o lato usa uno spazio $O(1)$, e le liste V e E usano uno spazio proporzionale al numero di elementi che contengono.

In termini di tempo d'esecuzione, la struttura a lista di lati fa quanto di meglio si possa fare quando si tratta di restituire il numero di vertici o di lati, così come quando deve generare un contenitore iterabile contenente i vertici o i lati: interrogando opportunamente le rispettive liste, V o E , i metodi `numVertices` e `numEdges` vengono eseguiti in un tempo $O(1)$, mentre scandendo le stesse liste i metodi `vertices` e `edges` producono il loro risultato in un tempo, rispettivamente, $O(n)$ e $O(m)$.

La limitazione più significativa di questa struttura, specialmente se confrontata con le altre rappresentazioni di grafo, è il tempo $O(m)$ che caratterizza l'esecuzione dei metodi `getEdge(u, v)`, `outDegree(v)` e `outGoingEdges(v)` (così come i corrispondenti metodi `inDegree` e `incomingEdges`). Il problema è che, dal momento che tutti i lati del grafo sono memorizzati nella lista non ordinata E , l'unico modo possibile per rispondere alle domande poste a questi metodi è quello di effettuare una scansione completa di tutti i lati.

Infine, prendiamo in esame i metodi che modificano la struttura del grafo. È facile aggiungere un nuovo vertice o un nuovo lato, in un tempo $O(1)$: ad esempio, per aggiungere un nuovo lato al grafo basta creare un esemplare di `Edge` che memorizzi come dato l'elemento fornito, aggiungendo poi tale oggetto di tipo "lato" alla lista posizionale E e memorizzando come attributo del lato stesso il riferimento di tipo `Position` ricevuto in risposta da E in seguito all'inserimento. Tale posizione, così memorizzata all'interno del lato, potrà essere utilizzata in seguito per individuare e rimuovere il lato stesso da E in un tempo $O(1)$, implementando così il metodo `removeEdge(e)`.

Merita un approfondimento il tempo d'esecuzione del metodo `removeVertex(v)`, che è $O(m)$. Come detto nella descrizione dell'ADT "grafo", quando si elimina dal grafo un vertice v , devono essere eliminati anche tutti i lati incidenti in v , altrimenti potrebbero, per assurdo, esistere nel grafo lati i cui vertici terminali non fanno parte del grafo stesso. Per individuare i lati incidenti nel vertice v , è necessario esaminare tutti i lati contenuti in E .

Tabella 14.2: Tempi d'esecuzione dei metodi di un grafo implementato con la struttura a lista di lati. Lo spazio occupato in memoria è $O(n + m)$, essendo n il numero di vertici e m il numero di lati.

Metodo	Tempo d'esecuzione
<code>numVertices()</code> , <code>numEdges()</code>	$O(1)$
<code>vertices()</code>	$O(n)$
<code>edges()</code>	$O(m)$
<code>getEdge(u, v)</code> , <code>outDegree(v)</code> , <code>outgoingEdges(v)</code>	$O(1)$
<code>insertVertex(x)</code> , <code>insertEdge(u, v, x)</code> , <code>removeEdge(e)</code>	$O(1)$
<code>removeVertex(v)</code>	$O(m)$

14.2.2 La struttura a lista di adiacenze (adjacency list)

La struttura a *lista di adiacenze* (*adjacency list*) per la rappresentazione di un grafo aggiunge alla struttura a lista di lati informazioni che consentono di accedere direttamente ai lati incidenti (e, quindi, ai vertici adiacenti) di ciascun vertice. Nello specifico, per ogni vertice v , gestiamo un contenitore $I(v)$, chiamato *insieme di incidenza* (*incidence collection*) di v , i cui elementi sono i lati incidenti in v . Nel caso di un grafo orientato, i lati entranti e uscenti sono memorizzati in due contenitori distinti, rispettivamente $I_{in}(v)$ e $I_{out}(v)$. Di solito l'insieme di incidenza $I(v)$ di un vertice v viene rappresentato con una lista ed è per questo motivo che questa rappresentazione di grafo viene detta "struttura a *lista di adiacenze*".

Richiediamo alla struttura principale di gestire l'insieme V di vertici in un modo che consente di individuare la struttura secondaria $I(v)$ associata a un dato vertice v in un tempo $O(1)$. Questo risultato si può ottenere usando una lista posizionale per rappresentare V , con ciascun esemplare di `Vertex` che contiene un riferimento diretto al proprio insieme di incidenza $I(v)$: questa struttura di rappresentazione di un grafo è illustrata nella Figura 14.5. Se i vertici vengono numerati in modo univoco da 0 a $n - 1$, si può usare, invece, una struttura principale basata su array per accedere alle liste secondarie.

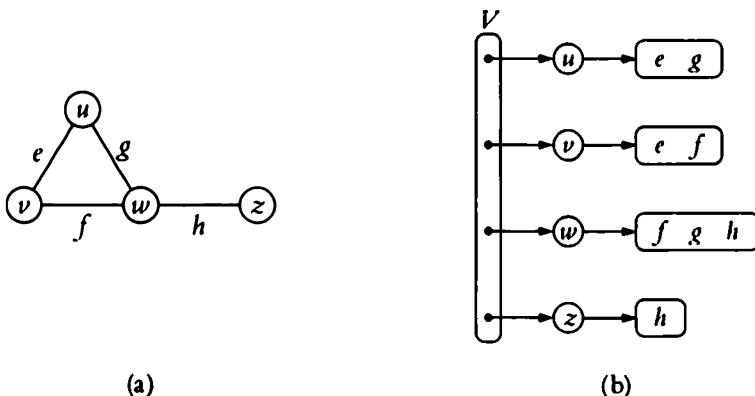


Figura 14.5: (a) Un grafo non orientato G ; (b) la rappresentazione schematica di G mediante la struttura a lista di adiacenze. L'insieme V è la lista di vertici principale e a ciascun vertice è associata una lista di lati incidenti. Anche se qui la situazione non è stata raffigurata in tal modo, immaginiamo che ciascun lato del grafo sia rappresentato da un esemplare di `Edge` che contiene riferimenti a propri vertici terminali, e che `E` sia una lista contenente tutti i lati.

Il beneficio principale di una struttura a lista di adiacenze è che l'insieme $I(v)$ (o, più specificatamente, $I_{\text{out}}(v)$), contiene esattamente quei lati che devono essere restituiti dal metodo $\text{outgoingEdges}(v)$, che, quindi, si può implementare facendo una scansione dei lati di $I(v)$, in un tempo $O(\deg(v))$, essendo $\deg(v)$ il grado del vertice v . Questa è la prestazione migliore che si può ottenere con qualunque rappresentazione di grafo, perché il metodo deve restituire un numero di lati uguale a $\deg(v)$.

Prestazioni della struttura a lista di adiacenze

La Tabella 14.3 riassume le prestazioni di un grafo implementato con una struttura a lista di adiacenze, nell'ipotesi che gli insiemi principali, V e E , e tutti gli insiemi secondari $I(v)$ siano realizzati con liste doppiamente concatenate.

Asintoticamente, lo spazio di memoria richiesto per una tale struttura è uguale a quello necessario per una struttura a lista di lati, cioè $O(n + m)$ per un grafo avente n vertici e m lati. Infatti, è evidente che le liste principali di vertici e lati utilizzano uno spazio $O(n + m)$. In aggiunta a questo, la somma delle lunghezze di tutte le liste secondarie è $O(m)$, per le stesse motivazioni che abbiamo utilizzato per dimostrare le Proposizioni 14.8 e 14.9: in sintesi, un lato non orientato (u, v) appartiene sia a $I(u)$ sia a $I(v)$, ma la sua presenza nel grafo produce un aumento di occupazione di spazio costante.

Abbiamo già osservato che il metodo $\text{outgoingEdges}(v)$ può essere realizzato in modo che abbia un tempo d'esecuzione $O(\deg(v))$ usando $I(v)$. Per un grafo orientato, questo risultato, più precisamente, diventa $O(\text{outdeg}(v))$, usando $I_{\text{out}}(v)$. Il metodo $\text{outDegree}(v)$ del grafo può essere realizzato in modo che venga eseguito in un tempo $O(1)$, se in tale tempo l'implementazione di $I(v)$ è in grado di fornire informazioni sulla propria dimensione. Per individuare un lato specifico, cioè per implementare $\text{getEdge}(u, v)$, possiamo fare una ricerca in $I(u)$ o in $I(v)$ (oppure, in un grafo orientato, in $I_{\text{out}}(u)$ o in $I_{\text{in}}(v)$): scegliendo l'insieme più piccolo, tra i due, otteniamo un tempo d'esecuzione $O(\min(\deg(u), \deg(v)))$.

Gli altri limiti riportati nella Tabella 14.3 si possono ottenere semplicemente facendo un po' di attenzione durante l'implementazione. Per consentire un'eliminazione efficiente dei lati, ciascun lato (u, v) deve conservare al proprio interno un riferimento alla propria posizione nelle liste $I(u)$ e $I(v)$, in modo da potervi essere cancellato in un tempo $O(1)$. Per eliminare un vertice v , dobbiamo anche eliminare tutti i suoi lati incidenti, però almeno ora siamo in grado di individuarli in un tempo $O(\deg(v))$.

Tabella 14.3: Tempi d'esecuzione dei metodi di un grafo implementato con la struttura a lista di adiacenze. Lo spazio occupato in memoria è $O(n + m)$, essendo n il numero di vertici e m il numero di lati.

Metodo	Tempo d'esecuzione
<code>numVertices()</code> , <code>numEdges()</code>	$O(1)$
<code>vertices()</code>	$O(n)$
<code>edges()</code>	$O(m)$
<code>getEdge(u, v)</code>	$O(\min(\deg(u), \deg(v)))$
<code>outDegree(v)</code> , <code>inDegree(v)</code>	$O(1)$
<code>outgoingEdges(v)</code> , <code>incomingEdges(v)</code>	$O(\deg(v))$
<code>insertVertex(x)</code> , <code>insertEdge(u, v, x)</code>	$O(1)$
<code>removeEdge(e)</code>	$O(1)$
<code>removeVertex(v)</code>	$O(\deg(v))$

14.2.3 La struttura a mappa di adiacenze (*adjacency map*)

Nella struttura a lista di adiacenze abbiamo ipotizzato che le strutture secondarie dedicate agli insiemi di adiacenza siano realizzate con liste concatenate non ordinate. In questo modo un insieme $I(v)$ usa uno spazio di memoria proporzionale a $O(\deg(v))$ e consente di inserire o rimuovere un lato in un tempo $O(1)$, mentre la scansione di tutti i lati incidenti in un vertice v avviene in un tempo $O(\deg(v))$. Tuttavia, la miglior implementazione possibile di $\text{getEdge}(u, v)$ richiede un tempo $O(\min(\deg(u), \deg(v)))$, perché bisogna fare una ricerca in $I(u)$ o in $I(v)$.

Possiamo migliorare quest'ultima prestazione temporale implementando, per ciascun vertice v , l'insieme $I(v)$ con una mappa basata su una tabella hash. In particolare, useremo come chiave nella mappa per ciascun lato il vertice terminale diverso da v , usando come valore proprio l'esemplare di "lato". Questa rappresentazione di grafo viene chiamata "struttura a *mappa di adiacenze*" (*adjacency map*) e la si può vedere schematizzata nella Figura 14.6. Lo spazio utilizzato per una struttura a mappa di adiacenze rimane $O(n + m)$, perché, per ciascun vertice v , $I(v)$ usa uno spazio $O(\deg(v))$, esattamente come avveniva con la struttura a lista di adiacenze.

Il vantaggio della mappa di adiacenze, rispetto alla lista di adiacenze, si evidenzia nel fatto che il metodo $\text{getEdge}(u, v)$ può essere realizzato in modo che il suo tempo d'esecuzione atteso sia $O(1)$, facendo una ricerca che usi il vertice u come chiave in $I(v)$, o viceversa. Questo costituisce un probabile miglioramento rispetto alla realizzazione mediante lista di adiacenze, anche se il caso peggiore rimane limitato da $O(\min(\deg(u), \deg(v)))$.

Confrontando le prestazioni della mappa di adiacenze con quelle delle altre rappresentazioni (come si può vedere nella Tabella 14.1), troviamo che questa raggiunge praticamente le prestazioni ottimali per tutti i metodi, costituendo così una scelta eccellente per la rappresentazione di grafi che non siano dedicati a compiti specifici.

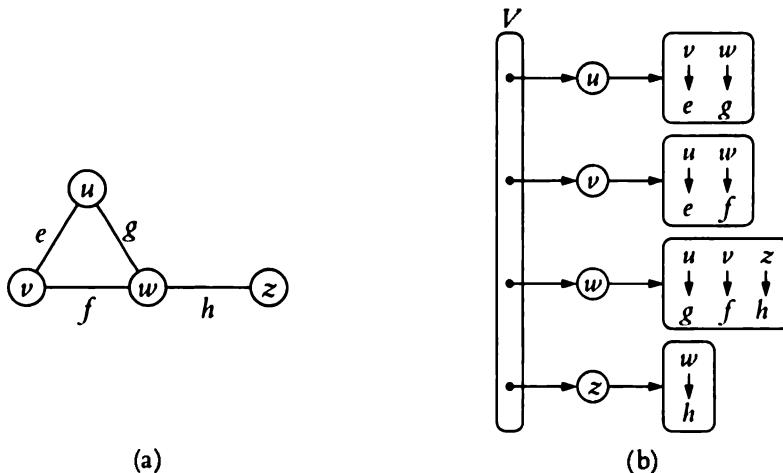


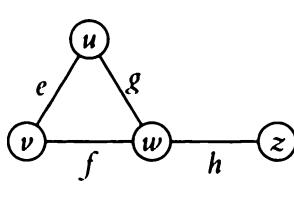
Figura 14.6: (a) Un grafo non orientato G ; (b) la rappresentazione schematica di G mediante la struttura a mappa di adiacenze. Ciascun vertice gestisce una propria mappa secondaria in cui i vertici adiacenti fungono da chiavi, associate ai lati corrispondenti usati come valori. Come nel caso della struttura a lista di adiacenze, ipotizziamo che sia anche presente una lista E contenente tutti gli esemplari di Edge che rappresentano lati del grafo.

14.2.4 La struttura a matrice di adiacenze (*adjacency matrix*)

La struttura a *matrice di adiacenze* (*adjacency matrix*) per la rappresentazione di un grafo G aggiunge alla lista di lati una matrice A (cioè un array bidimensionale, come visto nel Paragrafo 3.1.5), che consente di individuare in un tempo costante *nel caso peggiore* il lato (eventualmente) presente tra due vertici dati. In questa rappresentazione immaginiamo i due vertici come due numeri interi appartenenti all'insieme $\{0, 1, \dots, n - 1\}$, mentre i lati sono coppie di tali numeri interi. Questo ci consente di memorizzare i lati nelle celle di un'array A bidimensionale, $n \times n$: la cella $A[i][j]$ contiene un riferimento al lato (u, v) , se questo esiste, dove u è il vertice associato all'indice i e v è il vertice associato all'indice j . Se tale lato non è presente nel grafo, allora $A[i][j] = \text{null}$. Osserviamo che, se il grafo G è non orientato, l'array A è simmetrico, come si può vedere nella Figura 14.7, perché $A[i][j] = A[j][i]$ per ogni possibile coppia di valori i e j .

Il vantaggio più significativo della struttura a matrice di adiacenze è l'accesso in tempo costante, nel caso peggiore, a qualsiasi lato (u, v) , mentre dobbiamo ricordare che la struttura a mappa di adiacenze consente l'esecuzione di tale operazione in un tempo *atteso* costante. Tuttavia, alcune operazioni risultano essere meno efficienti quando eseguite con la matrice di adiacenze. Ad esempio, per trovare i lati incidenti al vertice v , dobbiamo esaminare tutte le n celle della riga associata a v , mentre possiamo ricordare che, con la lista di adiacenze o la mappa di adiacenze questo problema può essere risolto in un tempo ottimale, $O(\deg(v))$. L'aggiunta o la rimozione di vertici è, poi, particolarmente problematica, perché la matrice deve essere ridimensionata.

Inoltre, l'occupazione di spazio in memoria limitato da $O(n^2)$ è solitamente ben maggiore dello spazio $O(n + m)$ richiesto dalle altre rappresentazioni che abbiamo visto. Sebbene, nel caso peggiore, il numero di lati presenti in un grafo *denso* sarà proporzionale a n^2 , la maggior parte dei grafi usati da applicazioni realistiche sono *sparsi*: in tali casi, l'uso della matrice di adiacenze è inefficiente. Se, però, il grafo è denso, le costanti di proporzionalità tipiche della struttura a matrice di adiacenze possono essere minori di quelle delle strutture a lista di adiacenze o a mappa di adiacenze. Infatti, se i lati non contengono dati ausiliari, si può usare una matrice booleane di adiacenze, con un solo bit per ogni cella, in modo che $A[i][j] = \text{true}$ se e solo se (u, v) è un lato del grafo.



(a)

	0	1	2	3
$u \longrightarrow$	0	e	g	
$v \longrightarrow$	1	e	f	
$w \longrightarrow$	2	g	f	h
$z \longrightarrow$	3			h

(b)

Figura 14.7: (a) Un grafo non orientato G ; (b) la rappresentazione schematica di G mediante la struttura a matrice di adiacenze, nella quale i vertici sono messi in corrispondenza con i numeri interi da 0 a $n - 1$, usati come indici. Anche se qui la situazione non è stata raffigurata in tal modo, immaginiamo che ciascun lato del grafo sia rappresentato da un esemplare di Edge che contiene riferimenti ai propri vertici terminali. Ipotizziamo anche che sia presente, seppur non rappresentata qui, una lista secondaria contenente tutti i lati, in modo che il metodo edges() possa essere eseguito in un tempo $O(m)$ in un grafo avente m lati.

14.2.5 Implementazione in Java

In questo paragrafo presenteremo un'implementazione del tipo di dato astratto `Graph`, basata sulla rappresentazione mediante mappa di adiacenze, descritta nel Paragrafo 14.2.3. Per rappresentare le due liste principali, V e E , usiamo esemplari di liste posizionali, così come descritto nella presentazione della struttura a lista di lati. Inoltre, per ogni vertice v , usiamo una mappa basata su tabella hash per rappresentare l'insieme di incidenza $I(v)$, un contenitore secondario.

Per consentire la rappresentazione tanto di grafi orientati quanto di grafi non orientati in modo agevole, ogni vertice gestisce riferimenti a due diverse mappe: `outgoing` e `incoming`. In un grafo orientato questi due riferimenti sono inizializzati con due esemplari distinti di mappa, che rappresentano, rispettivamente, $I_{\text{out}}(v)$ e $I_{\text{in}}(v)$. Nel caso di grafo non orientato, invece, entrambi i riferimenti vengono inizializzati con l'indirizzo di un unico esemplare di mappa.

La nostra implementazione è organizzata in questo modo. Ipotizziamo che esistano le definizioni delle interfacce `Vertex`, `Edge` e `Graph`, così come le abbiamo descritte nel Paragrafo 14.1.1, anche se non le riportiamo qui per brevità. Poi, definiamo una classe concreta, `AdjacencyMapGraph`, con le classi annidate `InnerVertex` e `InnerEdge` che implementano le astrazioni di vertice e lato. Tutte queste classi sono parametriche e usano i parametri generici `V` e `E` per indicare i tipi degli elementi memorizzati, rispettivamente, nei vertici e nei lati.

Cominciamo con il Codice 14.1, che possiede le definizioni delle classi annidate (`InnerVertex` e `InnerEdge`) che, in realtà, devono essere inserite all'interno della definizione della classe `AdjacencyMapGraph`, che segue). Si osservi con attenzione come il costruttore di `InnerVertex` inizializzi le variabili di esemplare `outgoing` e `incoming` in base al fatto che il grafo sia complessivamente orientato oppure non orientato.

Il Codice 14.2 e il Codice 14.3 contengono il nucleo dell'implementazione della classe `AdjacencyMapGraph`. Un esemplare di grafo possiede una variabile booleana che indica se il grafo è orientato; inoltre, ha un riferimento alla lista dei suoi vertici e uno alla lista dei suoi lati. Anche se in questi frammenti di codice non sono stati riportati, la nostra implementazione prevede l'esistenza di due metodi privati `validate` che, dopo aver eseguito alcuni controlli di legittimità, effettuano le necessarie conversioni tra i tipi pubblici `Vertex` e `Edge` e i tipi privati concreti corrispondenti, `InnerVertex` e `InnerEdge`. Si tratta di uno schema progettuale simile a quello del metodo `validate` già inserito nella classe `LinkedPositionalList` (si veda il Codice 7.10 del Paragrafo 7.3.3), che converte un riferimento di tipo `Position`, proveniente dall'esterno, nel corrispondente esemplare di tipo `Node` di quella classe.

I metodi più complessi sono quelli che modificano la struttura del grafo. Quando viene invocato il metodo `insertVertex`, dobbiamo creare un nuovo esemplare di `InnerVertex` e aggiungerlo alla lista dei vertici, registrando al suo interno la sua posizione in tale lista (in modo da poterlo eliminare in modo efficiente dalla lista quando dovrà essere rimosso dal grafo). Quando inseriamo un lato, (u, v) , dobbiamo anche in questo caso creare un nuovo esemplare, aggiungerlo alla lista dei lati e registrare al suo interno la sua posizione in tale lista, tuttavia dobbiamo anche aggiungere il nuovo lato alla mappa delle adiacenze in uscita per il vertice u e alla mappa in ingresso per il vertice v . Il Codice 14.3 contiene anche il metodo `removeVertex`, mentre l'implementazione di `removeEdge` non è stata riportata per motivi di spazio.

Codice 14.1: Le classi `InnerVertex` e `InnerEdge` (da inserire all'interno della classe `AdjacencyMapGraph` come classi interne). Le interfacce `Vertex<V>` e `Edge<E>` non sono state riportate, per risparmiare spazio.

```

1  /** Un vertice nella rappresentazione di un grafo con mappa di adiacenze. */
2  private class InnerVertex<V> implements Vertex<V> {
3      private V element;
4      private Position<Vertex<V>> pos;
5      private Map<Vertex<V>, Edge<E>> outgoing, incoming;
6      /** Costruisce un nuovo esemplare di InnerVertex che memorizza l'elemento dato. */
7      public InnerVertex(V elem, boolean graphIsDirected) {
8          element = elem;
9          outgoing = new ProbeHashMap<>();
10         if (graphIsDirected)
11             incoming = new ProbeHashMap<>();
12         else
13             incoming = outgoing; // se non orientate, è solo un alias per outgoing
14     }
15     /** Restituisce l'elemento associato al vertice. */
16     public V getElement() { return element; }
17     /** Memorizza la posizione che ha questo vertice nella lista dei vertici. */
18     public void setPosition(Position<Vertex<V>> p) { pos = p; }
19     /** Restituisce la posizione che ha questo vertice nella lista dei vertici. */
20     public Position<Vertex<V>> getPosition() { return pos; }
21     /** Restituisce un riferimento alla mappa dei lati uscenti. */
22     public Map<Vertex<V>, Edge<E>> getOutgoing() { return outgoing; }
23     /** Restituisce un riferimento alla mappa dei lati entranti. */
24     public Map<Vertex<V>, Edge<E>> getIncoming() { return incoming; }
25 } // ----- fine della classe interna InnerVertex -----
26
27 /** Un lato tra due vertici. */
28 private class InnerEdge<E> implements Edge<E> {
29     private E element;
30     private Position<Edge<E>> pos;
31     private Vertex<V>[] endpoints;
32     /** Costruisce un esemplare di InnerEdge tra i vertici u e v. */
33     public InnerEdge(Vertex<V> u, Vertex<V> v, E elem) {
34         element = elem;
35         endpoints = (Vertex<V>[]) new Vertex[] {u, v}; // array di lunghezza 2
36     }
37     /** Restituisce l'elemento associato al lato. */
38     public E getElement() { return element; }
39     /** Restituisce un riferimento all'array dei vertici terminali. */
40     public Vertex<V>[] getEndpoints() { return endpoints; }
41     /** Memorizza la posizione che ha questo lato nella lista dei lati. */
42     public void setPosition(Position<Edge<E>> p) { pos = p; }
43     /** Restituisce la posizione che ha questo lato nella lista dei lati. */
44     public Position<Edge<E>> getPosition() { return pos; }
45 } // ----- fine della classe interna InnerEdge -----

```

Codice 14.2: Definizione della classe `AdjacencyMapGraph` (prosegue nel Codice 14.3). I metodi `validate(v)` e `validate(e)` non sono stati riportati.

```

1  public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
2      // le classi annidate InnerVertex e InnerEdge vanno inserite qui...
3      private boolean isDirected;
4      private PositionalList<Vertex<V>> vertices = new LinkedPositionalList<>();

```

```

5   private PositionalList<Edge<E>> edges = new LinkedPositionalList<>();
6   /** Costruisce un grafo vuoto (orientato oppure no). */
7   public AdjacencyMapGraph(boolean directed) { isDirected = directed; }
8   /** Restituisce il numero di vertici presenti nel grafo. */
9   public int numVertices() { return vertices.size(); }
10  /** Restituisce i vertici del grafo sotto forma di contenitore iterabile. */
11  public Iterable<Vertex<V>> vertices() { return vertices; }
12  /** Restituisce il numero di lati presenti nel grafo. */
13  public int numEdges() { return edges.size(); }
14  /** Restituisce i lati del grafo sotto forma di contenitore iterabile. */
15  public Iterable<Edge<E>> edges() { return edges; }
16  /** Restituisce il numero di lati che hanno origine nel vertice v. */
17  public int outDegree(Vertex<V> v) {
18      InnerVertex<V> vert = validate(v);
19      return vert.getOutgoing().size();
20  }
21  /** Restituisce un contenitore iterabile con i lati che hanno origine in v. */
22  public Iterable<Edge<E>> outgoingEdges(Vertex<V> v) {
23      InnerVertex<V> vert = validate(v);
24      return vert.getOutgoing().values(); // i lati sono i valori della mappa
25  }
26  /** Restituisce il numero di lati che hanno destinazione nel vertice v. */
27  public int inDegree(Vertex<V> v) {
28      InnerVertex<V> vert = validate(v);
29      return vert.getIncoming().size();
30  }
31  /** Restituisce un contenitore iterabile con i lati che hanno destinazione in v. */
32  public Iterable<Edge<E>> incomingEdges(Vertex<V> v) {
33      InnerVertex<V> vert = validate(v);
34      return vert.getIncoming().values(); // i lati sono i valori della mappa
35  }
36  /** Restituisce il lato che va da u a v; null se u e v non sono adiacenti. */
37  public Edge<E> getEdge(Vertex<V> u, Vertex<V> v) {
38      InnerVertex<V> origin = validate(u);
39      return origin.getOutgoing().get(v); // sarà null se non c'è lato da u a v
40  }
41  /** Restituisce i vertici del lato e come array di lunghezza due. */
42  public Vertex<V>[] endVertices(Edge<E> e) {
43      InnerEdge<E> edge = validate(e);
44      return edge.getEndpoints();
45  }

```

Codice 14.3: Definizione della classe `AdjacencyMapGraph` (continua dal Codice 14.2).
Il metodo `removeEdge` non è stato riportato.

```

46  /** Restituisce il vertice opposto a v lungo il lato e. */
47  public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
48                      throws IllegalArgumentException {
49      InnerEdge<E> edge = validate(e);
50      Vertex<V>[] endpoints = edge.getEndpoints();
51      if (endpoints[0] == v)
52          return endpoints[1];
53      else if (endpoints[1] == v)
54          return endpoints[0];
55      else
56          throw new IllegalArgumentException("v is not incident to this edge");
57  }

```

```

58  /** Inserisce e restituisce un nuovo vertice con l'elemento fornito. */
59  public Vertex<V> insertVertex(V element) {
60      InnerVertex<V> v = new InnerVertex<V>(element, isDirected);
61      v.setPosition(vertices.addLast(v));
62      return v;
63  }
64  /** Inserisce e restituisce un nuovo lato tra u e v, con l'elemento fornito. */
65  public Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E element)
66                      throws IllegalArgumentException {
67      if (getEdge(u, v) == null) {
68          InnerEdge<E> e = new InnerEdge<E>(u, v, element);
69          e.setPosition(edges.addLast(e));
70          InnerVertex<V> origin = validate(u);
71          InnerVertex<V> dest = validate(v);
72          origin.getOutgoing().put(v, e);
73          dest.getIncoming().put(u, e);
74          return e;
75      } else
76          throw new IllegalArgumentException("Edge from u to v exists");
77  }
78  /** Elimina dal grafo il vertice v e tutti i lati incidenti in v. */
79  public void removeVertex(Vertex<V> v) {
80      InnerVertex<V> vert = validate(v);
81      // elimina dal grafo tutti i lati incidenti in v
82      for (Edge<E> e : vert.getOutgoing().values())
83          removeEdge(e);
84      for (Edge<E> e : vert.getIncoming().values())
85          removeEdge(e);
86      // elimina v dalla lista di vertici
87      vertices.remove(vert.getPosition());
88  }
89 }

```

14.3 Attraversamenti di un grafo

La mitologia greca narra di un complesso labirinto che fu costruito per ospitare il mostruoso Minotauro, in parte toro e in parte uomo: un labirinto tanto complicato che nessuno poté uscirne, né un uomo né una bestia. Per meglio dire, nessun uomo riuscì a uscire da quel labirinto, almeno finché l'eroe greco Teseo, con l'aiuto della figlia del re, Arianna, non decise di utilizzare un algoritmo di *attraversamento di un grafo*. Teseo assicurò un rotolo di filo (che divenne poi famoso come "filo di Arianna") all'ingresso del labirinto, per svolgerlo passo dopo passo mentre attraversava i tortuosi passaggi in cerca del mostro. Teseo era ovviamente consapevole di aver progettato un buon algoritmo, perché, dopo aver trovato e sconfitto il mostro, seguì facilmente il filo per tornare sui propri passi fino all'entrata del labirinto, tra le braccia dell'amata Arianna.

Formalmente, un *attraversamento (traversal)* di un grafo è una procedura sistematica per esplorarlo, esaminandone tutti i vertici e tutti i lati. Un attraversamento è efficiente se visita tutti i vertici e tutti i lati in un tempo proporzionale al loro numero, cioè in un tempo lineare.

Gli algoritmi di attraversamento di un grafo sono strumenti chiave per rispondere a molte domande fondamentali relative ai grafi e, in particolare, al concetto di *raggiungibilità*, cioè per determinare come ci si possa spostare all'interno del grafo da un vertice a un altro.

seguendo percorsi interni al grafo stesso. Tra i problemi interessanti che hanno a che vedere con la raggiungibilità in un grafo non orientato G possiamo citare, ad esempio:

- Costruire un percorso dal vertice u al vertice v , oppure segnalare che tale percorso non esiste.
- Dato un vertice di partenza s in G , costruire, per ogni vertice v di G , un percorso da s a v avente il numero minimo di lati, oppure segnalare che tale percorso non esiste.
- Verificare se G è connesso.
- Costruire un albero ricoprente (*spanning tree*) di G , se G è connesso.
- Individuare i componenti connessi di G .
- Individuare un ciclo in G , oppure segnalare che G non ha cicli.

Tra i problemi interessanti che hanno a che vedere con la raggiungibilità in un grafo orientato \tilde{G} possiamo citare, ad esempio:

- Costruire un percorso orientato dal vertice u al vertice v , oppure segnalare che tale percorso non esiste.
- Trovare tutti i vertici di \tilde{G} che sono raggiungibili da un dato vertice s .
- Determinare se \tilde{G} è aciclico.
- Determinare se \tilde{G} è fortemente connesso.

Nel seguito di questo paragrafo presenteremo due efficienti algoritmi di attraversamento di un grafo, chiamati rispettivamente *attraversamento in profondità* (*depth-first search*) e *attraversamento in ampiezza* (*breadth-first search*).

14.3.1 Attraversamento in profondità (*depth-first search*)

Il primo algoritmo di attraversamento che analizziamo in questo paragrafo è l'*attraversamento in profondità* (*depth-first search*, DFS): è utile per verificare un gran numero di proprietà dei grafi, compresa la verifica di esistenza di un percorso che vada da un vertice a un altro e la verifica del fatto che un grafo sia connesso oppure no.

L'attraversamento in profondità di un grafo G è un'azione analoga all'attraversamento di un labirinto tenendo in mano un filo e una lattina di vernice, per non perderci. Partiamo da un vertice di partenza s in G , al quale assicuriamo un capo del nostro filo e che coloriamo per segnalare che l'abbiamo già "visitato". Il vertice s assume ora il ruolo di vertice "attuale" o "corrente". In generale, se indichiamo con u il nostro vertice attuale (che rappresenta la posizione in cui ci troviamo all'interno del grafo durante l'attraversamento), attraversiamo G prendendo in esame un lato qualsiasi (u, v) tra quelli incidenti nel vertice attuale u . Se il lato (u, v) ci porta a un vertice v che è già stato visitato (cioè colorato), ignoriamo tale lato; se, invece, (u, v) ci porta a un vertice v non ancora visitato, allora srotoliamo un po' di filo e andiamo in v . Quindi, coloriamo v per renderlo "visitato" e facciamo in modo che diventi il vertice "attuale", ripetendo l'elaborazione. Prima o poi giungeremo in un "vicolo cieco", cioè saremo in un vertice attuale v tale che tutti i lati incidenti in v portano a vertici già visitati. Per uscire da questa situazione di stallo, riavvolgiamo un po' di filo e torniamo sui nostri passi (un'azione che viene detta *backtracking*) lungo il lato che ci aveva portato in v , tornando a un vertice visitato in precedenza, u . A questo punto facciamo in modo che u

diventi il vertice attuale e ripetiamo l'elaborazione descritta per tutti i lati incidenti in u che non avevamo ancora preso in esame. Se tutti i lati incidenti in u ci portano in vertici già visitati, allora riavvolgiamo di nuovo un po' di filo e torniamo al vertice che ci aveva portato in u , ripetendo poi la procedura per quel vertice. Così facendo, continuiamo a tornare indietro lungo il percorso seguito in precedenza finché non troviamo un vertice che ha ancora qualche lato inesplorato, ne scegliamo uno e continuiamo l'attraversamento. La procedura termina quando, tornando sempre più indietro, arriviamo di nuovo al vertice di partenza s e scopriamo che non ci sono più lati inesplorati incidenti in s .

Il Codice 14.4 descrive, mediante pseudocodice, l'algoritmo di attraversamento in profondità a partire dal vertice u e segue passo dopo passo la nostra procedura, in analogia con l'uso di filo e colore. Usiamo la ricorsione per implementare il filo e ipotizziamo che esista un meccanismo (analogo alla verniciatura) per determinare se un vertice o un lato sono già stati esplorati in precedenza.

Codice 14.4: L'algoritmo DFS.

Algoritmo DFS(G, u):

Input: Un grafo G e un vertice u di G

Output: L'insieme di vertici raggiungibili da u , con i relativi lati di tipo "discovery" contrassegna il vertice u come visitato

for ogni lato $e = (u, v)$ uscente da u do

if il vertice v non è stato visitato then

contrassegna e come lato di tipo "discovery" per il vertice v

invoca ricorsivamente DFS(G, v)

Catalogare i lati del grafo con DFS

Si può usare un'esecuzione dell'attraversamento in profondità anche per analizzare la struttura di un grafo, sulla base del modo in cui ciascun lato viene esplorato durante l'attraversamento stesso. L'algoritmo DFS identifica in modo naturale quello che viene chiamato *albero di attraversamento in profondità* (o, più semplicemente, *albero DFS*) avente radice nel vertice di partenza s . Ogni volta che un lato, $e = (u, v)$, viene usato per scoprire un nuovo vertice v durante l'esecuzione dell'algoritmo DFS descritto nel Codice 14.4, tale lato diventa un *lato di tipo "discovery"* (cioè un lato "che ha scoperto" un nuovo vertice) o un *lato dell'albero DFS*, considerato come orientato da u a v . Tutti gli altri lati che vengono esaminati durante l'esecuzione di DFS vengono chiamati *nontree edges*, cioè "lati che non appartengono all'albero" DFS, e ci portano in un vertice già visitato in precedenza. Nel caso di grafo non orientato, scopriremo che tutti i lati che non appartengono all'albero DFS connettono il vertice attuale con un vertice che, nell'albero DFS, è un suo antenato: per questo motivo chiamiamo tali lati anche *lati back*, cioè lati "che tornano indietro". Quando, invece, eseguiamo l'algoritmo DFS su un grafo orientato, i lati che non appartengono all'albero DFS possono appartenere a tre diverse categorie:

- *lati back*, che connettono un vertice a un suo antenato nell'albero DFS
- *lati forward*, che connettono un vertice a un suo discendente nell'albero DFS
- *lati cross*, che connettono un vertice a un altro vertice che, nell'albero DFS, non ne è un antenato né un discendente

La Figura 14.8 mostra un esempio di applicazione dell'algoritmo DFS a un grafo orientato, evidenziando i diversi tipi di lati che non appartengono all'albero, mentre la Figura 14.9 mostra un esempio di applicazione dell'algoritmo DFS a un grafo non orientato.

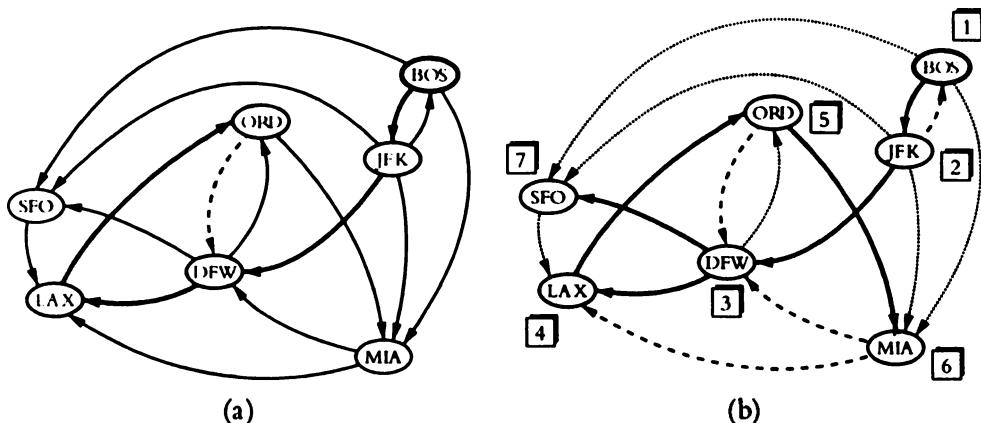


Figura 14.8: Un esempio di esecuzione dell'algoritmo DFS in un grafo orientato, partendo dal vertice (BOS): (a) una configurazione intermedia, dove, per la prima volta, il lato preso in esame porta a un vertice già visitato (DFW); (b) l'attraversamento DFS è stato completato. I lati dell'albero DFS sono rappresentati con linee di spessore maggiore, i lati *back* con linee tratteggiate e i lati *forward* e *cross* con linee punteggiate. L'ordine in cui i vertici sono stati visitati è indicato dalle etichette numerate poste di fianco a ciascun vertice. Il lato (ORD, DFW) è di tipo *back*, ma il lato (DFW, ORD) è di tipo *forward*. Il lato (BOS, SFO) è di tipo *forward* e il lato (SFO, LAX) è di tipo *cross*.

Proprietà dell'attraversamento in profondità

In merito all'algoritmo di attraversamento in profondità si possono fare parecchie osservazioni, molte delle quali sono conseguenza di come l'algoritmo DFS partiziona i lati di un grafo G in vari gruppi. Cominceremo con la proprietà più rilevante.

Proposizione 14.12: *Se G è un grafo non orientato in cui è stato eseguito l'attraversamento DFS a partire dal vertice iniziale s , allora l'attraversamento ha visitato tutti i vertici del componente连通 a cui appartiene s e i lati di tipo "discovery" formano un albero ricoprente di tale componente连通.*

Dimostrazione: Supponiamo, per assurdo, che esista almeno un vertice w che non sia stato visitato e che si trovi all'interno del componente连通 a cui appartiene s ; sia, poi, v il primo vertice non visitato lungo un qualche percorso che vada da s a w (che esiste senz'altro perché s e w appartengono allo stesso componente连通, come caso limite può essere $v = w$). Dato che v è il primo vertice non visitato lungo tale percorso, deve avere un vertice adiacente, u , che è stato visitato. Ma, quando l'algoritmo ha visitato u , deve aver preso in esame il lato (u, v) : quindi, il fatto che v sia rimasto "non visitato" non può essere vero. Di conseguenza, nel componente连通 a cui appartiene s non esistono vertici non visitati.

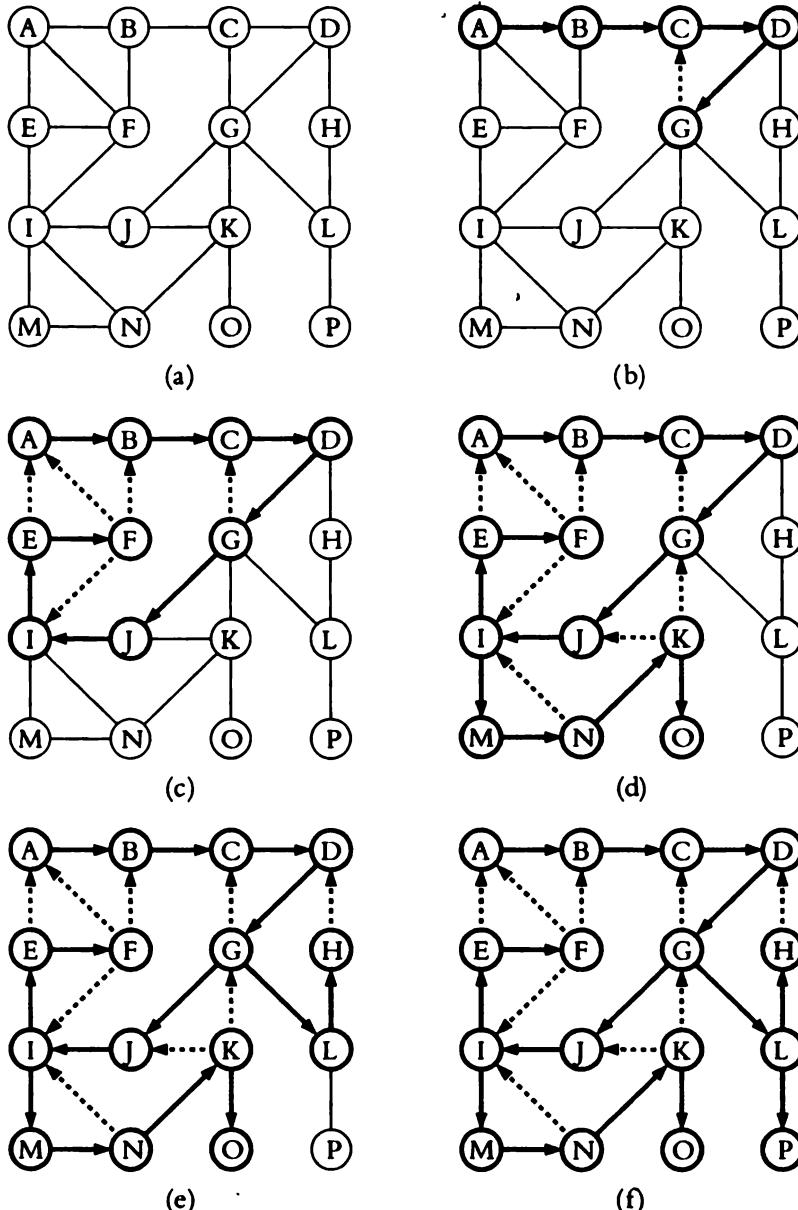


Figura 14.9: Esempio di esecuzione dell'attraversamento in profondità in un grafo non orientato, partendo dal vertice A, nell'ipotesi che le adiacenze di ciascun vertice siano esaminate in ordine alfabetico. Ad ogni passo intermedio sono stati evidenziati con spessore maggiore i vertici e i lati già esplorati, con i lati di tipo *discovery* disegnati a tratto continuo e i lati che non appartengono all'albero DFS (che sono di tipo *back*) disegnati con linee tratteggiate: (a) grafo in esame; (b) percorso di lati dell'albero DFS tracciato da A fino a quando non viene esaminato un lato *back*, (G, C); (c) l'attraversamento ha raggiunto F, che è un vicolo cieco; (d) dopo il ritorno in I (*backtracking*), si riparte con il lato (I, M) e si trova un altro vicolo cieco in O; (e) dopo il ritorno in G, si prosegue con il lato (G, L) e si trova un altro vicolo cieco in H; (f) risultato finale.

Dato che l'algoritmo segue un lato di tipo *discovery* soltanto quando si sposta in un vertice non visitato, l'insieme di tali lati non formerà mai un ciclo. Quindi, i lati *discovery* formano un sotto-grafo connesso privo di cicli, che è, per definizione, un albero. Inoltre, si tratta di un albero ricoprente, perché, come abbiamo già visto, l'attraversamento in profondità visita tutti i vertici del componente connesso a cui appartiene s . ■

Proposizione 14.13: Se \tilde{G} è un grafo orientato in cui è stato eseguito l'attraversamento DFS a partire dal vertice iniziale s , allora l'attraversamento ha visitato tutti i vertici di \tilde{G} che sono raggiungibili da s . Inoltre, l'albero DFS contiene percorsi orientati da s verso ciascun vertice raggiungibile da s .

Dimostrazione: Indichiamo con V , il sottoinsieme di vertici di \tilde{G} che sono stati visitati da DFS partendo dal vertice s . Vogliamo dimostrare che V , contiene s e che tutti i vertici raggiungibili da s appartengono a V . Supponiamo, per assurdo, che esista un vertice w raggiungibile da s che non appartiene a V . Consideriamo un percorso orientato da s a w e sia (u, v) il primo lato di tale percorso che esce da V , cioè tale che u appartiene a V , ma v non appartiene a V . Quando l'algoritmo DFS arriva nel vertice u , esplora tutti i lati uscenti da u e, quindi, deve raggiungere anche il vertice v , tramite il lato (u, v) . Di conseguenza, v appartiene a V , e siamo caduti in contraddizione. Quindi, V deve contenere tutti i vertici raggiungibili da s .

Dimostriamo ora la seconda affermazione per induzione, agendo sulle fasi dell'algoritmo. Affermiamo che, ogni volta che viene scoperto un lato (u, v) di tipo *discovery*, esiste un percorso orientato da s a v all'interno dell'albero DFS. Dato che u deve essere stato visitato in precedenza, esiste un percorso orientato da s a u , per cui, aggiungendo il lato (u, v) a tale percorso, otteniamo un percorso orientato da s a v . ■

Dal momento che i lati *back* collegano sempre un vertice v a un vertice u visitato in precedenza, osserviamo che ogni lato *back* implica la presenza di un ciclo in G , costituito dai lati di tipo *discovery* che vanno da u a v e dal lato *back*, (u, v) .

Tempo d'esecuzione dell'attraversamento in profondità

Per quanto riguarda il tempo d'esecuzione, l'algoritmo DFS è un metodo efficiente per attraversare un grafo. Osserviamo che DFS viene invocato al massimo una sola volta per ciascun vertice (dato che lo contrassegna come visitato) e, quindi, ogni lato viene esaminato al massimo due volte in un grafo non orientato (una volta a partire da ciascuno dei suoi due vertici terminali) e al massimo una sola volta in un grafo orientato (a partire dal suo vertice di origine). Se indichiamo con $n \leq n$ il numero di vertici raggiungibili da un vertice s e con $m_s \leq m$ il numero di lati incidenti in quei vertici, un'esecuzione di DFS iniziata in s richiede un tempo $O(n_s + m_s)$, se le condizioni seguenti sono soddisfatte:

- Il grafo è rappresentato con una struttura dati che consente, in un tempo $O(\deg(v))$, di costruire il contenitore restituito da *outgoingEdges*(v) e di effettuarne una scansione completa, così come consente di eseguire *opposite*(v, e) in un tempo costante.
- C'è la possibilità, in un tempo $O(1)$, di "contrassegnare" come esplorato un vertice o un lato e di verificare se un vertice o un lato è stato esplorato. Vedremo nel prossimo paragrafo come si possa implementare DFS per raggiungere questo obiettivo.

Soddisfatte le ipotesi appena enunciate, possiamo risolvere alcuni problemi interessanti.

Proposizione 14.14: Sia G un grafo non orientato con n vertici e m lati. L'attraversamento DFS di G può essere eseguito in un tempo $O(n + m)$ e può essere utilizzato per risolvere i problemi seguenti in un tempo $O(n + m)$:

- Costruire un percorso tra due vertici di G assegnati, se ne esiste uno.
- Verificare se G è连通的.
- Costruire un albero ricoprente di G , se G è连通的.
- Individuare i componenti连通的 di G .
- Individuare un ciclo in G , oppure segnalare che G non ha cicli.

Proposizione 14.15: Sia \tilde{G} un grafo orientato con n vertici e m lati. L'attraversamento DFS di \tilde{G} può essere eseguito in un tempo $O(n + m)$ e può essere utilizzato per risolvere i problemi seguenti in un tempo $O(n + m)$:

- Costruire un percorso orientato tra due vertici di \tilde{G} assegnati, se ne esiste uno.
- Costruire l'insieme dei vertici di \tilde{G} che sono raggiungibili da un vertice s assegnato.
- Verificare se \tilde{G} è fortemente连通的.
- Individuare un ciclo orientato in \tilde{G} , oppure segnalare che \tilde{G} è aciclico.

La dimostrazione delle Proposizioni 14.14 e 14.15 è basata su algoritmi che usano, come sotto-procedure, versioni lievemente modificate dell'algoritmo DFS. Nella parte restante di questo paragrafo ne esamineremo alcuni.

14.3.2 Implementazione di DFS e sue estensioni

Inizieremo il paragrafo con la presentazione di un'implementazione Java dell'algoritmo di attraversamento in profondità, che abbiamo già descritto con pseudocodice nel Codice 14.4. Per implementarlo, ci serve un meccanismo che tenga traccia di quali vertici sono stati visitati, oltre a memorizzare in qualche modo i lati che compongono l'albero DFS risultante. Per la gestione di queste informazioni useremo due strutture dati ausiliarie. Per prima cosa gestiremo un insieme, chiamato `known`, contenente i vertici che sono già stati visitati. Poi, usiamo una mappa, chiamata `forest`, che associa al vertice v il lato e del grafo che è stato usato per "scoprire" v (se ce n'è uno). Il Codice 14.5 presenta il nostro metodo DFS.

Codice 14.5: Implementazione ricorsiva dell'algoritmo di attraversamento in profondità di un grafo, a partire dal vertice assegnato u . Come risultato di un'invocazione, tutti i vertici visitati vengono aggiunti all'insieme dei vertici noti, `known`, mentre i lati di tipo `discovery` vengono aggiunti alla foresta `forest`.

```

1  /** Effettua l'attraversamento in profondità del grafo g a partire dal vertice u. */
2  public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
3      Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4      known.add(u);                                // il vertice u è stato scoperto
5      for (Edge<E> e : g.outgoingEdges(u)) { // per ogni lato uscente da u

```

```

6   Vertex<V> v = g.opposite(u, e);
7   if (!known.contains(v)) {
8       forest.put(v, e);           // e ha scoperto v, diventa lato dell'albero
9       DFS(g, v, known, forest); // esplora ricorsivamente a partire da v
10      }
11  }
12 }
```

Il nostro metodo `DFS` non fa nessuna ipotesi sull'implementazione degli esemplari di `Set` e `Map` che utilizza, ma l'analisi che vedremo nel paragrafo successivo, che porta a un tempo d'esecuzione $O(n + m)$, presuppone che si possa "contrassegnare" in qualche modo un vertice come "esplorato" o verificarne lo "stato di esplorazione" in un tempo $O(1)$. Se usiamo implementazioni mediante tabella hash dell'insieme e della mappa, allora tutte queste operazioni saranno eseguite in un tempo atteso $O(1)$ e l'intero algoritmo verrà eseguito in un tempo $O(n + m)$ con probabilità molto elevata. In pratica, si tratta di un compromesso che possiamo accettare.

Se i vertici possono essere etichettati con i numeri $0, \dots, n - 1$ (un'ipotesi che si fa molto spesso quando si progettano algoritmi che operano su grafi), allora l'insieme e la mappa possono essere implementati più direttamente con una tabella di ricerca (*lookup table*), usando l'etichetta di un vertice come indice in un array di dimensione n . In tal caso, le operazioni sull'insieme e sulla mappa che servono all'algoritmo possono essere eseguite in un tempo $O(1)$ nel caso peggiore. In alternativa, si può pensare di "decorare" ciascun vertice con l'informazione aggiuntiva, sfruttando la genericità del tipo dell'elemento che viene memorizzato in ciascun vertice oppure riprogettando il tipo `Vertex` in modo che contenga un campo aggiuntivo. Queste scelte progettuali consentono di eseguire in un tempo $O(1)$ tutte le operazioni che servono a contrassegnare i vertici come visitati e a controllare il loro relativo stato, senza ipotizzare che i vertici siano numerati.

Ricostruire un percorso da u a v

Possiamo usare il metodo `DFS` anche come strumento per individuare il percorso (orientato) che porta dal vertice u al vertice v , se v è raggiungibile da u . Tale percorso può essere facilmente ricostruito a partire dalle informazioni registrate nella foresta dei lati *discovery* durante l'attraversamento. Il Codice 14.6 mostra un'implementazione di un metodo ausiliario che genera una lista contenente, in ordine, i lati che si trovano lungo il percorso individuato da u a v , data la mappa di lati *discovery* prodotta dal metodo `DFS` descritto in precedenza.

Per ricostruire il percorso, iniziamo dalla fine del percorso stesso, esaminando la foresta di lati *discovery* per determinare quale sia il lato che è stato usato per raggiungere il vertice v . Poi, individuiamo il vertice opposto lungo tale lato e ripetiamo la procedura, determinando quale sia il lato della foresta che ha portato in quel vertice. Continuando in questo modo fino a raggiungere u possiamo ricostruire l'intero percorso. Ipotizzando che un accesso alla mappa avvenga in un tempo costante, la ricostruzione del percorso richiede un tempo proporzionale alla lunghezza del percorso stesso e, quindi, viene eseguito in un tempo $O(n)$, oltre ovviamente al tempo speso precedentemente per invocare `DFS`.

Codice 14.6: Metodo che ricostruisce un percorso orientato da u a v , nota che sia la mappa di lati *discovery* generata da un attraversamento in profondità iniziato dal vertice u . Il metodo restituisce una lista contenente, in ordine, i lati appartenenti al percorso.

```

1  /** Restituisce la lista di lati, in ordine, di un percorso orientato da u a v. */
2  public static <V,E> PositionalList<Edge<E>>
3      constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
4                      Map<Vertex<V>,Edge<E>> forest) {
5      PositionalList<Edge<E>> path = new LinkedPositionalList<>(){
6          if (forest.get(v) != null) {           // v è stato visitato dall'attraversamento
7              Vertex<V> walk = v;                // costruiamo il percorso a ritroso
8              while (walk != u) {
9                  Edge<E> edge = forest.get(walk);
10                 path.addFirst(edge);           // aggiunge edge **all'inizio** del percorso
11                 walk = g.opposite(walk, edge); // ripete dal vertice opposto
12             }
13         }
14     return path;
15 }
```

Verifica di connettività

Possiamo usare il metodo **DFS** anche per determinare se un grafo è connesso. Nel caso di grafo non orientato, basta semplicemente far partire l'attraversamento in profondità da un vertice qualsiasi e , alla fine, verificare se `known.size()` è uguale a n . Se il grafo è connesso, allora, per la Proposizione 14.12, tutti i suoi vertici saranno stati visitati; al contrario, se il grafo non è connesso, deve esistere almeno un vertice v che non è raggiungibile da u e che, quindi, non sarà stato visitato.

Se il grafo \tilde{G} è orientato, può darsi che ci interessi sapere se è **fortemente connesso**, cioè se, per ogni coppia di suoi vertici u e v , è vero che u è raggiungibile da v e che, viceversa, v è raggiungibile da u . Se facciamo partire un'invocazione di **DFS** indipendente da ciascun vertice, possiamo determinare se la condizione di connessione forte sia verificata, ma queste n invocazioni, combinate insieme, richiedono un tempo $O(n(n + m))$. In realtà si può determinare se \tilde{G} è fortemente connesso in un modo molto più rapido, che richiede soltanto due attraversamenti in profondità.

Partiamo con l'esecuzione di un attraversamento in profondità del nostro grafo orientato \tilde{G} a partire da un vertice qualsiasi, s . Se esiste almeno un vertice di \tilde{G} che non viene visitato da questo attraversamento e, quindi, non è raggiungibile da s , allora il grafo non è fortemente connesso. Se, invece, questo primo attraversamento visita tutti i vertici di \tilde{G} , allora dobbiamo verificare se s è raggiungibile da tutti gli altri vertici. In teoria, questo problema si può risolvere facendo una copia del grafo \tilde{G} , invertendo la direzione di tutti i suoi lati. Un attraversamento in profondità a partire da s del grafo invertito raggiungerà tutti i vertici da cui, nel grafo originario, è possibile raggiungere s . In pratica, invece di creare un nuovo grafo, è preferibile usare un'implementazione modificata del metodo **DFS** che esegua il suo ciclo principale per tutti i lati *entranti* nel vertice attuale, invece che per quelli *uscenti*. Dato che questo algoritmo esegue complessivamente soltanto due attraversamenti **DFS** di \tilde{G} , richiede un tempo $O(n + m)$.

Individuare tutti i componenti connessi

Quando un grafo non è connesso, può essere utile individuare tutti i *componenti connessi* del grafo, se questo non è orientato, oppure tutti i *componenti fortemente connessi*, se il grafo è orientato. Partiamo dal caso di un grafo non orientato.

Se una prima invocazione di `DFS` non riesce a raggiungere tutti i vertici del grafo, possiamo far partire una nuova invocazione di `DFS` da uno di tali vertici non ancora visitati. Il Codice 14.7 presenta una possibile implementazione di questo metodo complessivo, `DFSComplete`: restituisce una mappa che rappresenta una *foresta DFS* per l'intero grafo. Parliamo di foresta anziché di albero perché il grafo potrebbe non essere connesso.

I vertici che fungono da radici di alberi `DFS` in questa foresta non saranno associati ad alcun lato *discovery* e non compariranno come chiavi nella mappa che viene restituita, quindi il numero di componenti connessi del grafo g è uguale a $g.\text{numVertices}() - \text{forest}.\text{size}()$.

Codice 14.7: Metodo globale che restituisce una foresta `DFS` per un intero grafo.

```

1  /** Esegue DFS per un intero grafo e restituisce come mappa la foresta DFS. */
2  public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
3      Set<Vertex<V>> known = new HashSet<>();
4      Map<Vertex<V>,Edge<E>> forest = new HashMap<>();
5      for (Vertex<V> u : g.vertices())
6          if (!known.contains(u))
7              DFS(g, u, known, forest); // fa (ri)partire DFS da u
8      return forest;
9  }

```

Esaminando la struttura della foresta che viene restituita possiamo, poi, anche determinare quali vertici appartengono a ciascun componente, oppure possiamo apportare una piccola modifica al nucleo del metodo `DFS` in modo che etichetti ciascun vertice con il numero del componente a cui appartiene, nel momento in cui viene "scoperto" per la prima volta (si veda l'Esercizio C-14.43).

Sebbene il metodo `DFSComplete` invochi più volte il metodo `DFS` originario, il tempo totale richiesto per un'invocazione di `DFSComplete` è ancora $O(n + m)$. Nel caso di un grafo non orientato, ricordiamo, dalla nostra analisi originaria, che una singola invocazione di `DFS` a partire dal vertice s viene eseguita in un tempo $O(n_s + m_s)$, essendo n_s il numero di vertici raggiungibili da s e m_s il numero di lati incidenti in quei vertici. Dato che ogni invocazione di `DFS` esplora un componente diverso, la somma di tutti i termini di tipo $n_s + m_s$ è $n + m$.

Nel caso dell'individuazione dei componenti fortemente connessi di un grafo orientato, il problema è più complesso. Il limite superiore complessivo $O(n + m)$ per un'invocazione di `DFSComplete` si applica anche al caso di un grafo orientato, perché, quando si fa ripartire il metodo `DFS`, si procede usando l'insieme di vertici noti già esistente: questo garantisce che il metodo `DFS` sia invocato al massimo una volta per ciascun vertice e , quindi, che ogni lato uscente venga esplorato una sola volta durante l'intero processo.

Ad esempio, prendiamo nuovamente in esame il grafo della Figura 14.8. Se facessimo partire la prima invocazione del metodo `DFS` dal vertice `ORD`, l'insieme dei vertici noti diventerebbe uguale a { `ORD`, `DFW`, `SFO`, `LAX`, `MIA` }. Se, poi, facessimo ripartire il metodo `DFS` dal vertice `BOS`, i lati uscenti verso i vertici `SFO` e `MIA` non darebbero luogo a ulteriori invocazioni ricorsive, perché quei vertici appartengono già all'insieme dei vertici noti.

Tuttavia, la foresta restituita da un'unica invocazione di `DFSComplete` non rappresenta i componenti fortemente connessi del grafo. Esiste un approccio che individua tali compon-

nenti in un tempo $O(n + m)$, facendo uso di due invocazioni di `DFSComplete`, ma i dettagli di tale algoritmo vanno al di là degli obiettivi di questo libro.

Individuare cicli con DFS

Tanto nei grafi orientati quanto in quelli non orientati, esiste un ciclo se e solo se l'attraversamento DFS di quel grafo individua un *lato back*. È facile vedere che, se esiste un *lato back*, allora esiste un ciclo, che si può percorrere seguendo il *lato back* nella direzione che porta verso un antenato, per poi seguire i lati dell'albero fino al discendente di partenza. Viceversa, se nel grafo esiste un ciclo, allora deve esistere un *lato back* individuato durante l'esecuzione di DFS (anche se qui non dimostriamo questa seconda affermazione).

Dal punto di vista algoritmico, nel caso di un grafo non orientato individuare un *lato back* è semplice, perché tutti i lati che non fanno parte dell'albero DFS (cioè che non sono di tipo *discovery*) sono di tipo *back*. Nel caso di un grafo orientato, è necessario apportare qualche modifica all'implementazione del metodo `DFS` per poter assegnare la categoria giusta ai lati che non fanno parte dell'albero DFS, che possono essere di tipo *back*, *forward* o *cross*. Quando viene esplorato un lato orientato che porta a un vertice già visitato, dobbiamo capire se tale vertice sia un antenato di quello attuale nell'albero DFS che si va costruendo. Questo problema può essere risolto, ad esempio, usando un altro insieme contenente tutti i vertici per i quali è attiva un'invocazione ricorsiva di DFS: lasciamo l'implementazione come esercizio (C-14.42).

14.3.3 Attraversamento in ampiezza (*breadth-first search*)

Il meccanismo di avanzamento e di ritorno sui propri passi tipico dell'attraversamento in profondità, così come descritto nel paragrafo precedente, definisce un attraversamento che può essere effettuato fisicamente da una singola persona che esplori un grafo. In questo paragrafo, invece, esamineremo un altro algoritmo che attraversa un componente连通的 di un grafo, che prende il nome di *algoritmo di attraversamento in ampiezza (breadth-first search, BFS)*, ed è più adatto a spedire in esplorazione più persone, in tutte le direzioni, per attraversare collettivamente un grafo in modo coordinato.

Un attraversamento BFS procede per fasi successive e suddivide i vertici in *livelli*. Parte da un vertice, s , che viene posto al livello 0. Nella prima fase, colora come "visitati" tutti i vertici adiacenti al vertice di partenza s : questi vertici si trovano a un lato di distanza dal vertice di partenza e vengono posti al livello 1. Nella seconda fase, consentiamo a tutti gli esploratori di spostarsi di due passi (cioè di due lati) dal vertice di partenza: i nuovi vertici in cui giungono, che sono quelli adiacenti ai vertici di livello 1 e che non hanno ancora ricevuto l'assegnazione di un livello, vengono posti al livello 2 e colorati come "visitati". La procedura continua in modo analogo, terminando quando, esaminando un livello, non si trovano più nuovi vertici.

Il Codice 14.8 presenta un'implementazione di BFS in Java. Abbiamo seguito una strategia simile a quella del metodo `DFS` (presentato nel Codice 14.5), gestendo un insieme, `known`, dei vertici visitati e memorizzando i lati dell'albero BFS in una mappa. L'attraversamento BFS è illustrato dalla Figura 14.10.

Codice 14.8: Implementazione dell'algoritmo di attraversamento in ampiezza di un grafo, a partire dal vertice assegnato s .

```

1  /** Effettua l'attraversamento in ampiezza del grafo g a partire dal vertice s. */
2  public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
3      Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4      PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
5      known.add(s);
6      level.addLast(s);                                // il primo livello contiene solo s
7      while (!level.isEmpty()) {
8          PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
9          for (Vertex<V> u : level)
10             for (Edge<E> e : g.outgoingEdges(u)) {
11                 Vertex<V> v = g.opposite(u, e);
12                 if (!known.contains(v)) {
13                     known.add(v);
14                     forest.put(v, e);           // e ha scoperto v, diventa lato dell'albero
15                     nextLevel.addLast(v);     // v verrà esaminato al prossimo passo
16                 }
17             }
18         level = nextLevel; // il "prossimo" livello diventa il livello attuale
19     }
20 }
```

Parlando di DFS, abbiamo descritto una classificazione dei lati che non fanno parte dell'albero DFS come *lati back*, che connettono un vertice a un suo antenato (con riferimento, appunto, all'albero DFS), *lati forward*, che connettono un vertice a un suo discendente, e *lati cross*, che connettono un vertice a un altro vertice che non ne sia antenato né discendente. Nel BFS di un grafo non orientato, tutti i lati che non fanno parte dell'albero BFS sono lati di tipo *cross* (si veda l'Esercizio C-14.46), mentre nel BFS di un grafo orientato i lati che non fanno parte dell'albero possono essere di tipo *back* o *cross* (si veda l'Esercizio C-14.47).

L'algoritmo di attraversamento BFS ha diverse proprietà interessanti, alcune delle quali saranno messe in evidenza nelle proposizioni che seguono. La cosa forse più interessante è che un percorso interno all'albero BFS che vada dalla radice s a un altro vertice qualsiasi, v , è quello di lunghezza minima, in termini di numero di lati, tra tutti i possibili percorsi che, nel grafo, vanno da s a v .

Proposizione 14.16: Se G è un grafo (orientato o non orientato) in cui è stato eseguito l'attraversamento BFS a partire dal vertice s , allora:

- L'attraversamento ha visitato tutti i vertici di G che sono raggiungibili da s .
- Per ogni vertice v a livello i , il percorso che, nell'albero BFS T , va da s a v ha un numero di lati uguale a i e qualsiasi altro percorso che, in G , va da s a v ha almeno i lati.
- Se (u, v) è un lato che non appartiene all'albero BFS, allora il livello di v può essere al massimo superiore di un'unità del livello di u .

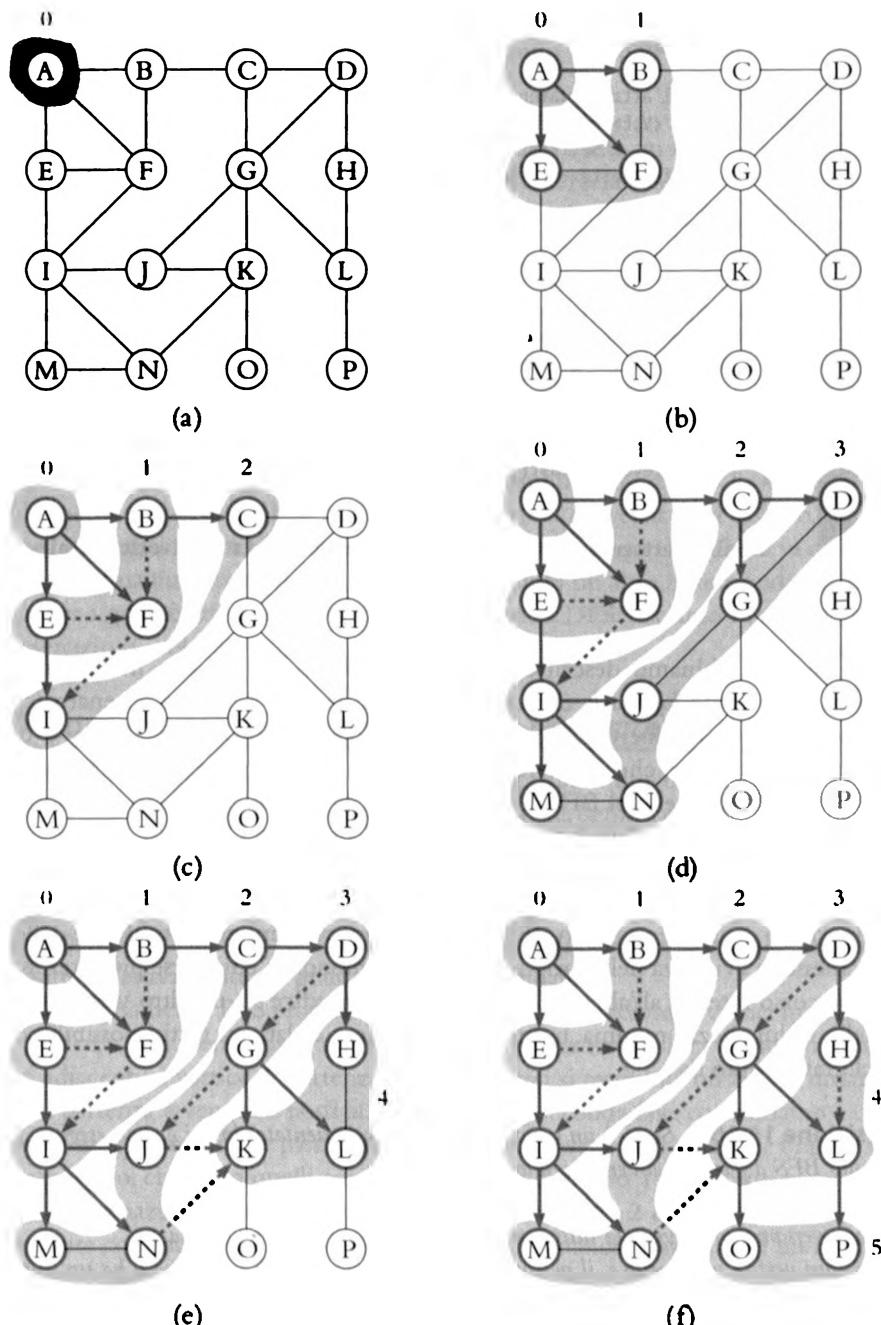


Figura 14.10: Esempio di esecuzione dell'attraversamento in ampiezza, nell'ipotesi che le adiacenze di ciascun vertice siano esaminate in ordine alfabetico. I lati di tipo *discovery* sono disegnati con linee a tratto continuo, mentre i lati (di tipo *cross*) che non fanno parte dell'albero BFS sono disegnati con linee tratteggiate: (a) la ricerca parte dal vertice A; (b) visita dei vertici del livello 1; (c) visita dei vertici del livello 2; (d) visita dei vertici del livello 3; (e) visita dei vertici del livello 4; (f) visita dei vertici del livello 5.

La dimostrazione di queste affermazioni è lasciata all'Esercizio C-14.49.

L'analisi del tempo d'esecuzione richiesto dall'attraversamento BFS è simile a quella vista per l'attraversamento DFS e porta a un tempo $O(n + m)$ o, più specificatamente, $O(n + m)$, dove n , è il numero di vertici raggiungibili dal vertice s e m , $\leq m$ è il numero di lati incidenti in tali vertici. Per esplorare l'intero grafo, si può far ripartire la procedura da un altro vertice, in modo analogo a quanto fatto nel metodo `DFSCcomplete`, visto nel Codice 14.7. Il metodo `constructPath`, descritto nel Codice 14.6, è in grado di ricostruire il percorso più breve che va dal vertice s al vertice v .

Proposizione 14.17: *Se G è un grafo avente n vertici e m lati rappresentato con la struttura a lista di adiacenze, l'attraversamento BFS di G richiede un tempo $O(n + m)$.*

Anche se la nostra implementazione di BFS, presentata nel Codice 14.8, opera livello per livello, l'algoritmo BFS si può anche implementare usando un'unica coda FIFO per rappresentare il confine della zona già raggiunta dalla ricerca. Si inizia inserendo nella coda il vertice di partenza, estraendo ripetutamente il vertice che si trova all'inizio della coda e inserendo alla fine della coda i suoi vertici adiacenti che non siano ancora stati visitati (si veda l'Esercizio C-14.50).

Confrontando le funzionalità di DFS e BFS, si vede che entrambi gli algoritmi possono essere utilizzati per trovare in modo efficiente l'insieme dei vertici che sono raggiungibili a partire da un vertice assegnato, così come per determinare percorsi verso tali vertici. Tuttavia, l'algoritmo BFS garantisce che quei percorsi comprendano il numero minimo di lati. In un grafo non orientato, entrambi gli algoritmi possono essere utilizzati per verificare se il grafo è连通的, per individuare i suoi componenti连通的 e per identificare un ciclo al suo interno. In un grafo orientato, l'algoritmo DFS è più adatto a risolvere alcuni problemi, come la ricerca di un ciclo orientato o l'individuazione dei suoi componenti fortemente连通的.

14.4 Chiusura transitiva

Abbiamo visto che gli attraversamenti di un grafo possono essere utilizzati per rispondere a domande fondamentali relative al problema della raggiungibilità in un grafo orientato. In particolare, se siamo interessati a sapere se esiste un percorso che vada dal vertice u al vertice v , possiamo eseguire un attraversamento DFS o BFS a partire da u e osservare se v viene visitato. Se rappresentiamo il grafo con una struttura a lista di adiacenze o a mappa di adiacenze, possiamo rispondere a questa domanda sulla raggiungibilità di v da u in un tempo $O(n + m)$, come garantito dalle Proposizioni 14.15 e 14.17.

In alcune applicazioni possiamo aver bisogno di rispondere a molte domande di raggiungibilità in modo più efficiente, nel qual caso può essere utile predisporre una versione più adeguata della rappresentazione del grafo. Ad esempio, la prima fase di un servizio che calcoli un percorso di guida per andare da un punto a un altro di una mappa stradale può essere la verifica che la destinazione sia raggiungibile. Analogamente, in una rete elettrica, possiamo voler essere in grado di determinare velocemente se la corrente sta fluendo regolarmente da un particolare vertice a un altro. Spinti da questi esempi di applicazioni, presentiamo la definizione seguente. La chiusura transitiva (transitive closure) di un grafo

orientato \tilde{G} è un grafo orientato \tilde{G}' che ha gli stessi vertici di \tilde{G} e ha un lato (u, v) se e solo se \tilde{G} ha un percorso orientato che va da u a v (compreso, ovviamente, il caso in cui (u, v) sia un lato di \tilde{G}).

Se un grafo è rappresentato mediante una struttura a lista di adiacenze o a mappa di adiacenze, siamo in grado di calcolare la sua chiusura transitiva in un tempo $O(n(n + m))$ facendo uso di n attraversamenti del grafo, usando ogni volta come vertice di partenza un diverso vertice del grafo \tilde{G} . Ad esempio, un attraversamento DFS che parta dal vertice u è in grado di determinare l'insieme dei vertici raggiungibili da u e, quindi, l'insieme dei lati che, nella chiusura transitiva, hanno origine in u .

Nella parte restante di questo paragrafo presenteremo una tecnica alternativa per costruire la chiusura transitiva di un grafo orientato che è particolarmente adatta al caso in cui il grafo sia rappresentato mediante una struttura che consente di eseguire il metodo `getEdge(u, v)` in un tempo costante (ad esempio, la struttura a matrice di adiacenze). Sia \tilde{G} un grafo orientato avente n vertici e m lati. Calcoliamo la sua chiusura transitiva procedendo per fasi successive, partendo con l'inizializzazione $\tilde{G}_0 = \tilde{G}$. Sempre all'inizio, numeriamo i vertici di \tilde{G} in modo arbitrario da 1 a n : v_1, v_2, \dots, v_n . Poi, iniziamo la fase 1. Nella generica fase k , costruiamo il grafo orientato \tilde{G}_k partendo con l'assegnazione $\tilde{G}_k = \tilde{G}_{k-1}$ e aggiungendo a \tilde{G}_k il lato orientato (v_i, v_j) se e solo se il grafo orientato \tilde{G}_{k-1} contiene entrambi i lati, (v_i, v_k) e (v_k, v_j) . In questo modo, garantiamo che venga rispettata la semplice regola contenuta nella proposizione che segue.

Proposizione 14.18: Per $i = 1, \dots, n$, il grafo orientato \tilde{G}_k contiene il lato (v_i, v_j) se e solo se il grafo orientato \tilde{G} contiene un percorso orientato che va da v_i a v_j , i cui vertici intermedi (se ve ne sono) appartengono all'insieme $\{v_1, \dots, v_k\}$. In particolare, \tilde{G}_n coincide con \tilde{G}' , la chiusura transitiva di \tilde{G} .

La Proposizione 14.18 suggerisce un semplice algoritmo che è in grado di calcolare la chiusura transitiva di \tilde{G} ed è basato su una sequenza di fasi, durante ciascuna delle quali calcola uno dei grafi \tilde{G}_k . L'algoritmo è noto come *algoritmo di Floyd-Warshall* e il suo pseudocodice è presentato nel Codice 14.9, mentre la Figura 14.11 mostra un esempio della sua esecuzione.

Codice 14.9: Pseudocodice che descrive l'algoritmo di Floyd-Warshall, che calcola la chiusura transitiva \tilde{G}' di \tilde{G} mediante la costruzione incrementale di una sequenza di grafi orientati $\tilde{G}_0, \tilde{G}_1, \dots, \tilde{G}_n$ con $k = 1, \dots, n$.

Algoritmo FloydWarshall(\tilde{G}):

Input: Un grafo orientato \tilde{G} con n vertici

Output: La chiusura transitiva \tilde{G}' di \tilde{G}

sia v_1, v_2, \dots, v_n una qualsiasi numerazione dei vertici di \tilde{G}

$\tilde{G}_0 = \tilde{G}$

for k che va da 1 a n do

$\tilde{G}_k = \tilde{G}_{k-1}$

for ogni coppia i, j , con i e j in $\{1, \dots, n\}$, $i \neq j$, $i \neq k, j \neq k$ do

if i due lati (v_i, v_k) e (v_k, v_j) sono presenti in \tilde{G}_{k-1} then

aggiungi il lato (v_i, v_j) a \tilde{G}_k (se non c'è già)

return \tilde{G}_n

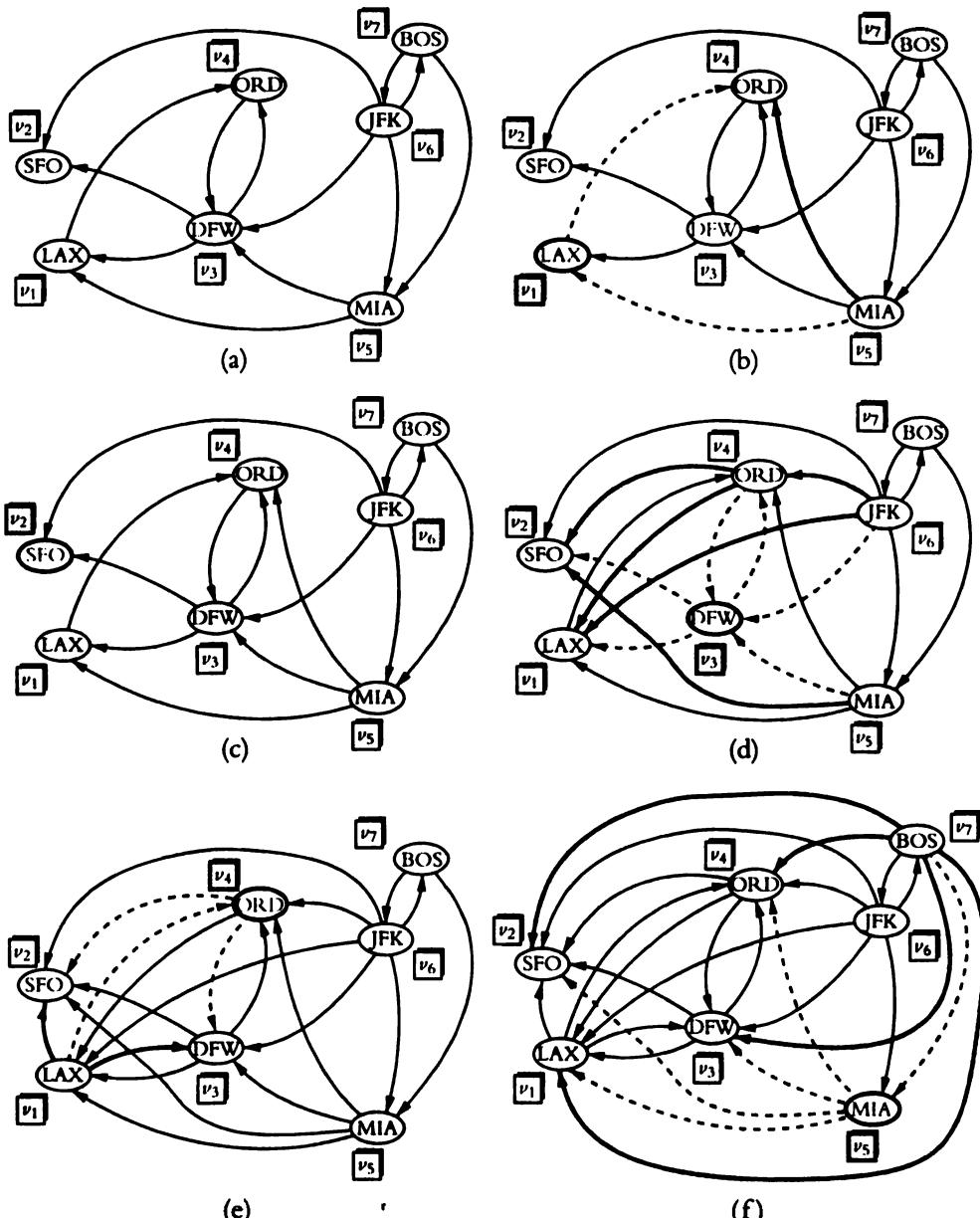


Figura 14.11: Sequenza di grafi orientati calcolati dall'algoritmo di Floyd-Warshall: (a) grafo orientato iniziale, $\tilde{G} = \tilde{G}_0$, e numerazione dei vertici; (b) grafo orientato \tilde{G}_1 ; (c) \tilde{G}_2 ; (d) \tilde{G}_3 ; (e) \tilde{G}_4 ; (f) \tilde{G}_5 . Si osservi che $\tilde{G}_5 = \tilde{G}_6 = \tilde{G}_7$. Se il grafo orientato \tilde{G}_{k-1} contiene i lati (v_i, v_k) e (v_k, v_j) , ma non il lato (v_i, v_j) , nel disegno del grafo \tilde{G}_k mostriamo i lati (v_i, v_k) e (v_k, v_j) con linee tratteggiate e il lato (v_i, v_j) con linea di spessore maggiore. Ad esempio, in (b) esistono i lati (MIA, LAX) e (LAX, ORD), che generano il lato (MIA, ORD).

Tramite questo pseudocodice, possiamo facilmente analizzare il tempo d'esecuzione dell'algoritmo Floyd-Warshall ipotizzando che la struttura dati che rappresenta \tilde{G} consenta

l'esecuzione dei metodi `getEdge` e `insertEdge` in un tempo $O(1)$. Il ciclo principale viene eseguito n volte, mentre il ciclo più interno prende in esame ciascuna delle $O(n^2)$ coppie di vertici, eseguendo per ognuna di esse una quantità di elaborazione che richiede un tempo costante. Di conseguenza, il tempo d'esecuzione complessivo dell'algoritmo di Floyd-Warshall è $O(n^3)$. Dalla descrizione e dall'analisi appena condotta, deriva la proposizione seguente.

Proposizione 14.19: *Sia \tilde{G} un grafo orientato con n vertici, rappresentato mediante una struttura dati che consente la ricerca e la modifica di informazioni di adiacenza in un tempo $O(1)$. In questo caso, l'algoritmo di Floyd-Warshall calcola la chiusura transitiva \tilde{G}^* di \tilde{G} in un tempo $O(n^3)$.*

Prestazioni dall'algoritmo di Floyd-Warshall

Le prestazioni temporali asintotiche $O(n^3)$ dell'algoritmo di Floyd-Warshall non sono migliori di quelle che si potrebbero ottenere eseguendo ripetutamente DFS, una volta per ciascun vertice, per calcolarne l'insieme di raggiungibilità. In effetti, quando il grafo è denso oppure quando è sparso ma è rappresentato mediante la matrice di adiacenze, l'algoritmo di Floyd-Warshall ottiene prestazioni che asintoticamente coincidono con quelle del DFS ripetuto (si veda l'Esercizio R-14.13).

L'algoritmo di Floyd-Warshall è interessante soprattutto perché è di più facile implementazione rispetto al DFS ripetuto e, in pratica, è anche molto più veloce, perché, "nascosto" nella notazione asintotica, c'è un numero di operazioni elementari molto inferiore. L'algoritmo è particolarmente adatto all'utilizzo di una matrice di adiacenze, dove si può usare un singolo bit per indicare la raggiungibilità di v da u , che viene modellata nella chiusura transitiva da un lato (u, v) .

Si noti, comunque, che le invocazioni ripetute di DFS danno luogo complessivamente a prestazioni asintoticamente migliori quando il grafo è sparso e viene rappresentato usando una struttura a lista di adiacenze o a mappa di adiacenze. In tal caso, una singola esecuzione di DFS richiede un tempo $O(n + m)$ per cui la chiusura transitiva viene calcolata in un tempo $O(n^2 + nm)$, che è preferibile rispetto a $O(n^3)$.

Implementazione in Java

Concluderemo con un'implementazione dell'algoritmo di Floyd-Warshall in Java, presentata nel Codice 14.10. Sebbene lo pseudocodice dell'algoritmo descriva una sequenza di grafi orientati, $\tilde{G}_0, \tilde{G}_1, \dots, \tilde{G}_n$, in realtà il nostro metodo modifica direttamente il grafo originario, aggiungendo ripetutamente nuovi lati alla chiusura mentre vengono eseguite le fasi successive dell'algoritmo di Floyd-Warshall.

Ancora, lo pseudocodice dell'algoritmo descrive i cicli che scandiscono i vertici usando indici che vanno da 0 a $n - 1$, ma, nel nostro ADT "grafo", preferiamo usare la sintassi Java specifica per i cicli `for-each`, operando direttamente sui vertici del grafo. Quindi, nel Codice 14.10, le variabili i, j e k , sono, in realtà, riferimenti a vertici, non indici interi da usare nella sequenza di vertici.

Infine, nella nostra implementazione Java usiamo un'ulteriore ottimizzazione, in confronto allo pseudocodice, evitando di effettuare la scansione dei valori di j a meno che non abbiano verificato che il lato (i, k) esista effettivamente nell'attuale versione della chiusura.

Codice 14.10: Implementazione dell'algoritmo di Floyd-Warshall in Java.

```

1  /** Converte il grafo g nella sua chiusura transitiva. */
2  public static <V,E> void transitiveClosure(Graph<V,E> g) {
3      for (Vertex<V> k : g.vertices())
4          for (Vertex<V> i : g.vertices())
5              // verifica se il lato (i,k) esiste nell'attuale chiusura parziale
6              if (i != k && g.getEdge(i,k) != null)
7                  for (Vertex<V> j : g.vertices())
8                      // verifica se il lato (k,j) esiste nell'attuale chiusura parziale
9                      if (i != j && j != k && g.getEdge(k,j) != null)
10                         // se (i,j) non c'è ancora, aggiungilo alla chiusura
11                         if (g.getEdge(i,j) == null)
12                             g.insertEdge(i, j, null);
13 }

```

14.6 Grafi orientati aciclici

I grafi orientati privi di cicli orientati si riscontrano in molte applicazioni e spesso vengono chiamati **grafi orientati aciclici o DAG** (*directed acyclic graph*), usando, per brevità, l'acronimo inglese. Tra le applicazioni di grafi di questo tipo, citiamo:

- Prerequisiti tra insegnamenti in un piano di studi accademico.
- Ereditarietà tra classi in un programma orientato agli oggetti.
- Vincoli di pianificazione (*scheduling*) tra parti di un progetto.

Per approfondire questo ultimo tipo di applicazione, useremo l'esempio seguente:

Proposizione 14.20: *Per poter gestire un progetto di grandi dimensioni, è utile scomporlo in un insieme di compiti più limitati. Raramente, però, i singoli compiti sono indipendenti tra loro, per cui esistono dei vincoli di pianificazione che li coinvolgono reciprocamente (ad esempio, nel progetto di costruzione di un edificio, il compito che prevede di acquistare i chiodi deve chiaramente precedere quello che inchioda le tegole sul tetto). Ovviamente i vincoli di pianificazione non possono essere caratterizzati da circolarità, perché questo renderebbe il progetto impossibile (ad esempio, per trovare lavoro dovete avere esperienza, ma per avere esperienza lavorativa dovete aver avuto un lavoro, una serie di vincoli chiaramente circolare). I vincoli di pianificazione impongono delle restrizioni all'ordine in cui i singoli compiti possono essere eseguiti: se un vincolo dice che il compito A deve essere terminato prima che il compito B possa iniziare, allora nell'ordine d'esecuzione dei compiti A deve precedere B. Di conseguenza, se rappresentiamo un insieme di compiti che può essere pianificato correttamente (cioè che non presenta circolarità nei vincoli) usandolo come insieme di vertici di un grafo orientato e inseriamo in tale grafo un lato orientato da u a v ogni volta che un vincolo stabilisce che il compito u debba essere eseguito prima del compito v, allora questo porta alla definizione di un grafo orientato aciclico.*

14.5.1 Ordinamento topologico

L'esempio precedente ci spinge a dare la definizione seguente. Sia \tilde{G} un grafo orientato con n vertici. Un ordinamento topologico (topological ordering) di \tilde{G} è una disposizione v_1, v_2, \dots, v_n dei vertici di \tilde{G} tale che, per ogni lato (v_i, v_j) di \tilde{G} , si ha $i < j$. In pratica, un ordinamento topologico è una disposizione dei suoi vertici tale che qualunque percorso orientato di \tilde{G} contenga vertici disposti seguendo la disposizione stessa, da sinistra a destra. Come si può vedere nella Figura 14.12, un grafo orientato può avere più ordinamenti topologici distinti.

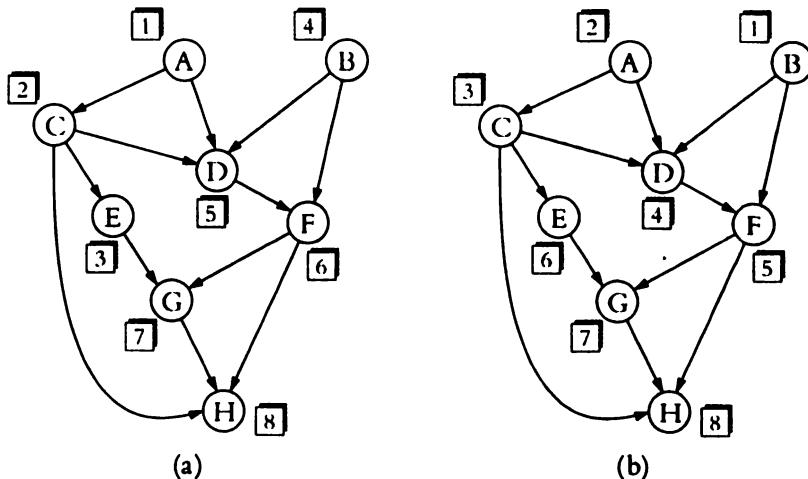


Figura 14.12: Due ordinamenti topologici dello stesso grafo orientato aciclico.

Proposizione 14.21: Il grafo orientato \tilde{G} ha un ordinamento topologico se e solo se è aciclico.

Dimostrazione: La necessità dell'affermazione (cioè la parte “solo se” dell'enunciato) si dimostra facilmente. Supponiamo che \tilde{G} abbia un ordinamento topologico e ipotizziamo, per assurdo, che contenga un ciclo costituito dai lati $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_{k-1}}, v_{i_0})$. Per l'ordinamento topologico, deve essere $i_0 < i_1 < \dots < i_{k-1} < i_0$, che, evidentemente, è impossibile. Quindi, \tilde{G} deve essere aciclico.

Occupiamoci ora della sufficienza dell'affermazione (cioè della parte “se”). Supponiamo che \tilde{G} sia aciclico: troveremo un algoritmo che costruisce un ordinamento topologico per \tilde{G} . Dato che \tilde{G} è aciclico, deve avere un vertice privo di lati entranti (cioè con grado entrante uguale a zero): sia v_1 tale vertice. In effetti, se v_1 non esistesse, allora seguendo un percorso orientato a partire da un vertice arbitrario incontreremmo prima o poi un vertice già appartenente al percorso stesso, che sarebbe quindi un ciclo, contraddicendo il fatto che \tilde{G} sia aciclico. Se eliminiamo da \tilde{G} il vertice v_1 e tutti i suoi lati uscenti, il grafo risultante è ovviamente ancora aciclico, quindi anch'esso ha un vertice privo di lati entranti, che chiamiamo v_2 . Ripetendo questa procedura finché il grafo non rimane vuoto, otteniamo una disposizione v_1, v_2, \dots, v_n dei vertici di \tilde{G} . Per costruzione, se (v_i, v_j) è un lato di \tilde{G} , allora v_i

deve essere stato eliminato prima di v_i , e, quindi, deve essere $i < j$. Di conseguenza, v_1, v_2, \dots, v_n è un ordinamento topologico di G . ■

La dimostrazione della Proposizione 14.21 suggerisce un algoritmo per generare un ordinamento topologico di un grafo orientato, che chiamiamo proprio *algoritmo di ordinamento topologico (topological sorting)*. Nel Codice 14.11 presentiamo un'implementazione di questa tecnica in Java, mentre la Figura 14.13 illustra un esempio di esecuzione dell'algoritmo. La nostra implementazione usa una mappa, `inCount`, per mettere in corrispondenza ciascun vertice v con un contatore che rappresenta il numero aggiornato di lati entranti in v , escludendo quelli che provengono da vertici che sono stati aggiunti in precedenza all'ordinamento topologico che si sta costruendo. Come nel caso dei nostri algoritmi di attraversamento di un grafo, l'uso di una mappa basata su tabella hash consente di effettuare accessi in un tempo atteso costante, ma non nel caso peggiore: quest'ultima condizione si potrebbe ottenere facilmente se i vertici potessero essere numerati da 0 a $n - 1$, oppure se memorizzassimo il conteggio come campo aggiuntivo degli oggetti che rappresentano i singoli vertici.

Come "effetto collaterale", l'algoritmo di ordinamento topologico descritto nel Codice 14.11 verifica anche se il grafo orientato G su cui opera è aciclico. Infatti, se l'algoritmo termina senza aver inserito tutti i vertici nell'ordinamento, allora il sotto-grafo contenente i vertici che non sono stati ordinati deve contenere almeno un ciclo orientato.

Codice 14.11: Implementazione dell'algoritmo di ordinamento topologico in Java (nella Figura 14.13 si può vedere un esempio di esecuzione).

```

1  /** Restituisce una lista dei vertici del DAG g in ordine topologico. */
2  public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
3      // lista dei vertici in ordine topologico
4      PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
5      // contenitore dei vertici che non hanno più vincoli
6      Stack<Vertex<V>> ready = new LinkedStack<>();
7      // mappa che tiene traccia del grado entrante residuo di ciascun vertice
8      Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
9      for (Vertex<V> u : g.vertices()) {
10          inCount.put(u, g.inDegree(u)); // inizializza con il grado entrante effettivo
11          if (inCount.get(u) == 0)        // se u non ha lati entranti,
12              ready.push(u);           // allora è libero da vincoli
13      }
14      while (!ready.isEmpty()) {
15          Vertex<V> u = ready.pop();
16          topo.addLast(u);
17          for (Edge<E> e : g.outgoingEdges(u)) { // considera tutti i vicini uscenti da u
18              Vertex<V> v = g.opposite(u, e);
19              inCount.put(v, inCount.get(v) - 1); // senza u, v ha un vincolo in meno
20              if (inCount.get(v) == 0)
21                  ready.push(v);
22          }
23      }
24  }
25 }
```

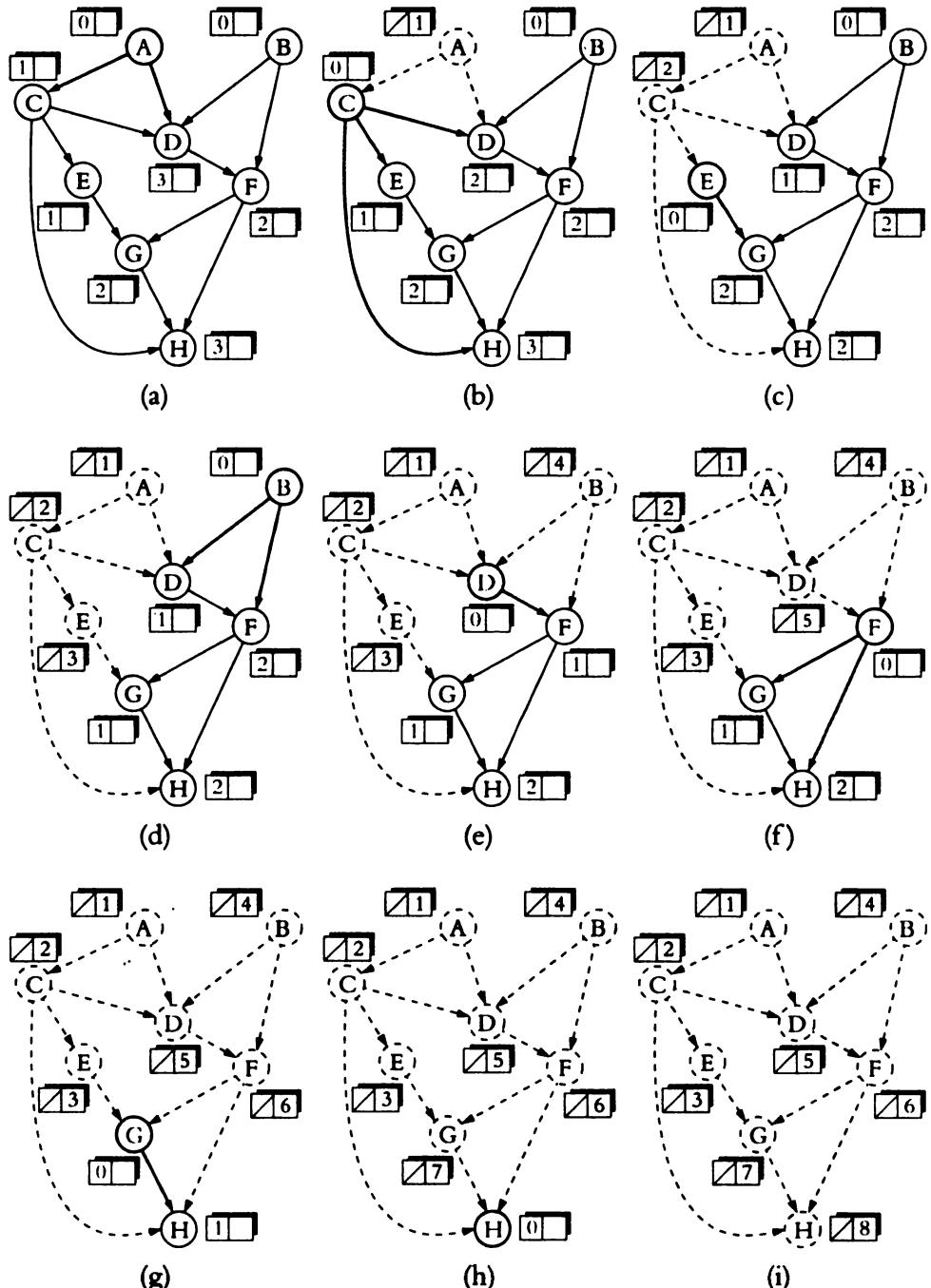


Figura 14.13: Esempio di esecuzione dell’algoritmo `topologicalSort` (descritto nel Codice 14.11). L’etichetta vicina a ciascun vertice mostra il suo valore aggiornato nella mappa `inCount` e la sua posizione nell’ordinamento che si sta costruendo, se è già definita. Il vertice di volta in volta evidenziato è quello avente conteggio uguale a zero e selezionato per diventare il successivo vertice nell’ordinamento topologico. Le linee tratteggiate denotano lati che sono già stati esaminati e che, quindi, non influiscono più sui valori memorizzati nella mappa `inCount`.

Proposizione 14.22: Sia \tilde{G} un grafo orientato con n vertici e m lati, rappresentato con una struttura a lista di adiacenze. L'algoritmo di ordinamento topologico viene eseguito in un tempo $O(n + m)$ usando uno spazio di memoria ausiliario $O(n)$ e costruisce un ordinamento topologico di \tilde{G} oppure non riesce a includervi alcuni vertici, segnalando così il fatto che \tilde{G} contiene almeno un ciclo orientato.

Dimostrazione: La memorizzazione iniziale degli n gradi entranti richiede un tempo $O(n)$, perché il metodo `inDegree` è tempo-costante. Diciamo che un vertice u è stato *visitato* dall'algoritmo di ordinamento topologico quando u viene rimosso dalla lista `ready`. Un vertice u può essere visitato soltanto quando `inCount.get(u)` restituisce 0, cosa che implica che siano stati visitati tutti i suoi predecessori, cioè i vertici che hanno lati uscenti che arrivano in u . Di conseguenza, qualunque vertice che appartenga a un ciclo orientato non verrà mai visitato, mentre qualunque altro vertice verrà visitato una e una sola volta. L'algoritmo scandisce una sola volta tutti i lati uscenti da ciascun vertice visitato, per cui il tempo d'esecuzione è proporzionale al numero di lati uscenti dai vertici visitati. In base alla Proposizione 14.9, il tempo d'esecuzione è, quindi, $O(n + m)$. Per quanto riguarda lo spazio di memoria utilizzato, osserviamo che i contenitori `topo`, `ready` e `inCount` hanno al massimo una voce per ciascun vertice, quindi usano uno spazio $O(n)$. ■

14.6 Ricerca dei percorsi più brevi

Come visto nel Paragrafo 14.3.3, si può usare la strategia di attraversamento in ampiezza (BFS) per trovare un percorso che vada da un vertice iniziale a un qualsiasi altro vertice di un grafo connesso e che abbia il minimo numero di lati possibile. Questo approccio ha senso in tutti quei casi in cui i vertici sono tra loro equivalenti ai fini dei percorsi, ma esistono molte situazioni in cui questo approccio non è adeguato.

Ad esempio, potremmo voler utilizzare un grafo per rappresentare le strade di collegamento tra varie città e potremmo essere interessati a trovare la via più veloce per viaggiare attraverso il paese. In questo caso, probabilmente non è giusto considerare tutti i lati come equivalenti, perché alcune distanze tra due città possono essere molto maggiori di altre. Analogamente, potremmo usare un grafo per rappresentare una rete di calcolatori (come Internet) e ci potrebbe interessare il fatto di scoprire il percorso più veloce per trasferire un pacchetto di dati da un computer a un altro. Anche in questo caso non sarà probabilmente appropriato che tutti i lati vengano considerati tra loro equivalenti, perché tipicamente alcune connessioni tra calcolatori di una rete sono molto più veloci di altre (ad esempio, alcuni lati del grafo possono rappresentare connessioni con ampiezza di banda molto limitata, mentre altri possono rappresentare connessioni in fibra ottica ad altissima velocità). È naturale, quindi, considerare grafi i cui lati non abbiano tutti lo stesso "peso".

14.6.1 Grafi pesati

Un **grafo pesato** (*weighted graph*) è un grafo che ha un'etichetta numerica (ad esempio, intera) $w(e)$ associata a ciascun lato e e chiamata **peso** (*weight*) del lato e . Essendo il lato $e = (u, v)$, usiamo anche la notazione $w(e) = w(u, v)$. La Figura 14.14 mostra un esempio di grafo pesato.

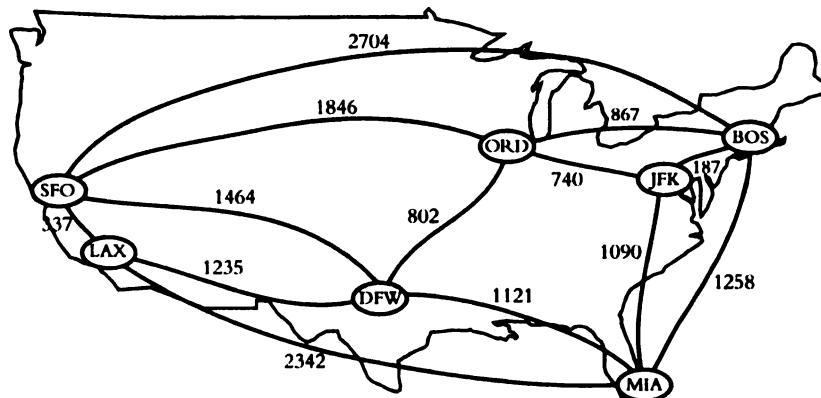


Figura 14.14: Un grafo pesato nel quale i vertici rappresentano i principali aeroporti statunitensi e i pesi dei lati rappresentano distanze, in miglia. Questo grafo ha un percorso da JFK a LAX di peso totale 2777 (passando per ORD e DFW), che è anche il percorso di peso minimo che va da JFK a LAX.

Definizione dei percorsi più brevi in un grafo pesato

Sia G un grafo pesato. La *lunghezza* (o il *peso*) di un percorso P è la somma dei pesi dei lati di P . Se $P = ((v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k))$, allora la lunghezza di P , indicata con $w(P)$, è così definita:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

La *distanza* di un vertice u da un vertice v in G , indicata con $d(u, v)$, è la lunghezza del percorso di lunghezza minima (detto anche *il percorso più breve, shortest path*) tra tutti quelli che vanno da u a v , se ne esiste almeno uno.

Quando in G non esiste un percorso che va da u a v , spesso si scrive, per convenzione, che $d(u, v) = \infty$. Anche se esiste un percorso da u a v in G , se in G c'è anche un ciclo il cui peso totale è negativo, la distanza da u a v può non essere definita. Ad esempio, supponiamo che i vertici di G rappresentino città e che i pesi dei lati di G rappresentino il costo, in denaro, per andare da una città all'altra. Se qualcuno volesse pagarcì per andare da JFK a ORD, allora il "costo" del lato (JFK, ORD) sarebbe negativo. Se, poi, qualcun altro volesse pagarcì per andare da ORD a JFK, allora in G sarebbe presente un ciclo di peso negativo e le distanze (nel senso di costi) non sarebbero più definite: chiunque potrebbe, infatti, costruire un percorso (contenente cicli) in G per andare da qualsiasi città A a qualsiasi altra città B che prima passi per JFK e poi continui a percorrere il ciclo tra JFK e ORD tutte le volte che vuole, prima di andare finalmente in B . L'esistenza di tale ciclo ci consentirebbe di costruire percorsi con costo negativo a piacere (guadagnando anche una fortuna!). Ma le distanze non possono essere numeri negativi a piacere: quindi, quando usiamo pesi dei lati di un grafo per rappresentare distanze, dobbiamo fare attenzione a non introdurre nel grafo cicli di peso negativo.

Supponiamo che ci sia dato un grafo pesato G e che ci venga chiesto di trovare un percorso di peso minimo (*shortest path*) partendo da un vertice s e arrivando in qualsiasi

altro vertice in G , vedendo i pesi dei lati come distanze. In questo paragrafo vedremo modi efficienti per trovare tutti questi percorsi di peso minimo, se ne esistono. Il primo algoritmo che presentiamo si applica al caso, semplice ma frequente, in cui tutti i pesi dei lati di G sono non negativi, cioè $w(e) \geq 0$ per ogni lato e di G (per cui sappiamo a priori che in G non esistono cicli con peso negativo). Ricordiamo anche che il caso speciale di calcolo di un percorso di peso minimo quando tutti i pesi sono uguali a uno può essere risolto con l'algoritmo di attraversamento BFS, presentato nel Paragrafo 14.3.3.

C'è un approccio molto interessante per risolvere questo problema che riguarda una sola origine per i percorsi cercati (*single-source shortest paths*) e sfrutta lo schema progettuale del *metodo greedy*, visto nel Paragrafo 13.4.2. Ricordiamo che, usando questo schema progettuale, risolviamo un problema facendo ripetutamente la scelta migliore tra quelle disponibili in ciascuna iterazione. Questo paradigma può spesso essere utilizzato in situazioni dove cerchiamo di ottimizzare una funzione di costo definita per un insieme di oggetti: possiamo aggiungere oggetti all'insieme, uno alla volta, scegliendo sempre il successivo che ottimizza la funzione tra quelli che non sono ancora stati scelti.

14.6.2 L'algoritmo di Dijkstra

L'idea principale su cui si basa l'applicazione del metodo greedy al problema del percorso più breve da una sola origine è quella di eseguire un attraversamento in ampiezza "pesato" a partire dal vertice s che rappresenta l'origine dei percorsi cercati. In particolare, possiamo usare il metodo greedy per sviluppare un algoritmo che faccia crescere, iterazione dopo iterazione, una "nuvola" (*cloud*) di vertici attorno a s , con i vertici che entrano nella nuvola in ordine crescente della loro distanza da s . Quindi, in ciascuna iterazione dell'algoritmo, il vertice successivo che viene scelto è quello, tra i vertici esterni alla nuvola, che si trova più vicino a s . L'algoritmo termina quando non ci sono più vertici fuori dalla nuvola (oppure quando i vertici esterni alla nuvola non sono connessi a quelli interni): in quel momento disponiamo di un percorso di peso minimo da s verso ogni vertice di G che sia raggiungibile da s . Questo approccio è un semplice ma potente esempio dello schema progettuale secondo il metodo greedy. L'applicazione del metodo greedy al problema del percorso più breve da una sola origine produce un algoritmo che è noto come *algoritmo di Dijkstra*.

Riassamento dei lati

Definiamo un'etichetta $D[v]$ per tiascun vertice v in V , che useremo per approssimare la distanza da s a v all'interno del grafo G . Il significato di queste etichette è che $D[v]$ conterrà sempre la lunghezza del miglior percorso da s a v che abbiamo trovato *fino a quel momento*. Inizialmente, $D[s] = 0$ e $D[v] = \infty$ per ogni $v \neq s$; poi, definiamo l'insieme C , inizialmente vuoto, che sarà la nostra "nuvola" di vertici. Ad ogni iterazione dell'algoritmo, selezioniamo un vertice u che non sia in C e che abbia l'etichetta minima, $D[u]$, per poi inserirlo in C (in generale, useremo una coda prioritaria per scegliere un vertice tra quelli esterni alla nuvola). Durante la prima iterazione inseriremo, ovviamente, s in C . Ogni volta che un nuovo vertice, u , viene inserito in C , aggiorniamo l'etichetta $D[v]$ di ogni vertice v che sia adiacente a u e che sia ancora al di fuori di C , in modo che tenga conto del fatto che ci può essere un percorso nuovo e migliore per arrivare in v da s passando per u . Questa

operazione di aggiornamento viene chiamata “procedura di *rilassamento*” (*relaxation*), perché prende un vecchio valore approssimato e controlla se può essere migliorato per avvicinarsi di più al proprio valore vero. Più in dettaglio, l’operazione di rilassamento di un lato può essere così descritta:

Rilassamento di un lato

```
if  $D[u] + w(u, v) < D[v]$  then  
   $D[v] = D[u] + w(u, v)$ 
```

Descrizione dell’algoritmo e un esempio

Nel Codice 14.12 presentiamo lo pseudocodice per l’algoritmo di Dijkstra, mentre un’illustrazione di alcune sue iterazioni viene proposta nelle Figure 14.15, 14.16 e 14.17.

Codice 14.12: Pseudocodice che descrive l’algoritmo di Dijkstra, che risolve il problema di trovare percorsi di peso minimo da una sola origine per un grafo orientato o non orientato.

Algoritmo ShortestPath(G, s):

Input: Un grafo G , orientato o non orientato, con pesi non negativi nei lati, e un suo vertice, s

Output: La lunghezza del percorso più breve da s a v , per ogni vertice v di G
inizializza $D[s] = 0$ e $D[v] = \infty$ per ogni vertice $v \neq s$

crea una coda prioritaria Q con tutti i vertici di G usando le etichette D come chiavi
while Q non è vuota do

```
{ inserisci un nuovo vertice nella nuvola }  
 $u$  = valore restituito da  $Q.\text{removeMin}()$   
for ogni lato  $(u, v)$  tale che  $v$  appartenga a  $Q$  do  
  { esegui la procedura di rilassamento per il lato  $(u, v)$  }  
  if  $D[u] + w(u, v) < D[v]$  then  
     $D[v] = D[u] + w(u, v)$   
    cambia la chiave associata a  $v$  in  $Q$ : diventa  $D[v]$   
return l’etichetta  $D[v]$  di ciascun vertice  $v$ 
```

Perché funziona

L’aspetto forse più interessante dell’algoritmo di Dijkstra è che, nel momento in cui un vertice u viene inserito nella nuvola C , la sua etichetta $D[u]$ contiene proprio la lunghezza corretta del percorso più breve che va da v a u . Quindi, quando l’algoritmo termina, ha calcolato la lunghezza del percorso più breve (cioè, per definizione, la distanza) da s a ogni altro vertice di G , cioè ha risolto il problema del percorso più breve da una sola origine (*single-source shortest paths*).

È probabile che non sia così evidente il motivo per cui l’algoritmo di Dijkstra calcola correttamente la distanza dal vertice di partenza s di tutti gli altri vertici u del grafo. Perché, nel momento in cui il vertice u viene estratto dalla coda prioritaria Q e aggiunto alla nuvola C , il valore dell’etichetta $D[u]$ è proprio uguale alla distanza di s da u ? La risposta a questa domanda dipende dal fatto che nel grafo non sono presenti lati di peso negativo: questa ipotesi consente al metodo greedy di funzionare correttamente, come dimostriamo nell’affermazione che segue.

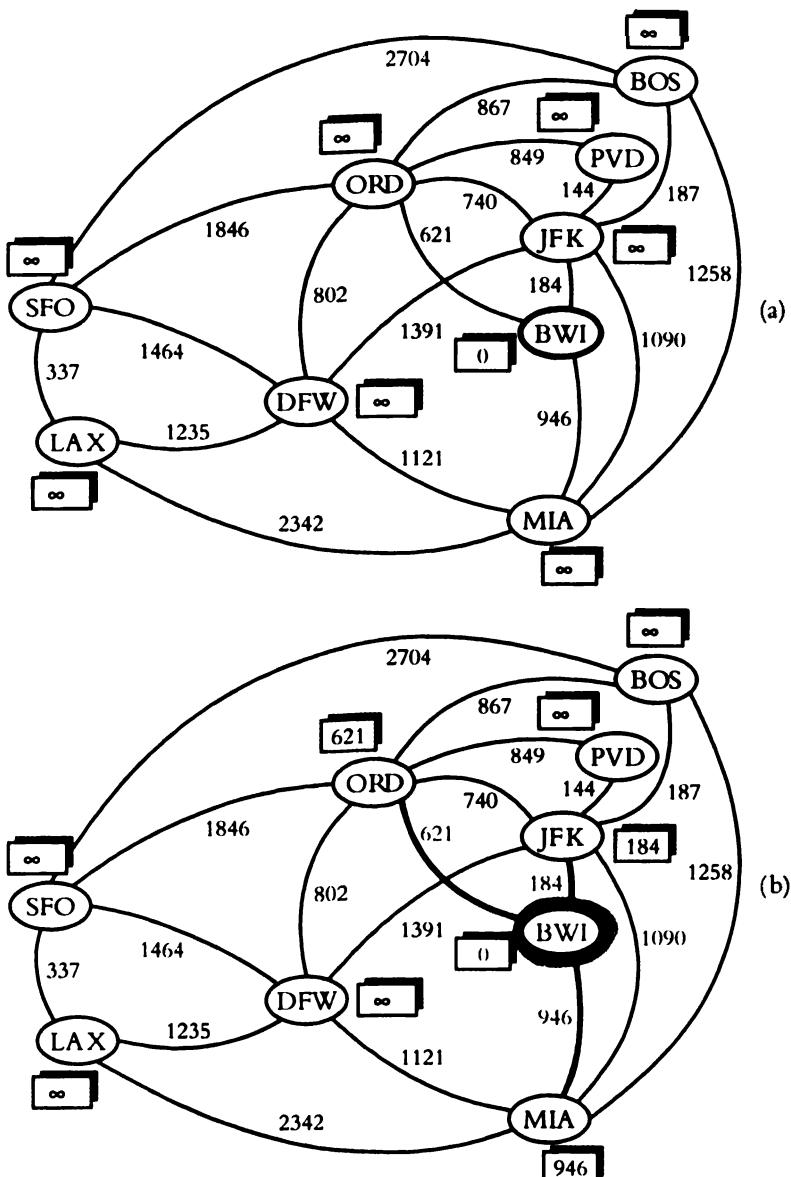


Figura 14.15: Un esempio di esecuzione dell'algoritmo di Dijkstra per la ricerca della lunghezza del percorso più breve in un grafo pesato. Il vertice di partenza è BWI. Il rettangolo posto accanto a ciascun vertice v contiene la sua etichetta, $D[v]$. I lati dell'albero dei percorsi più brevi (shortest-path tree) saranno disegnati con una freccia di spessore maggiore. Per ogni vertice u esterno alla "nuvola" viene mostrato con un tratto di spessore maggiore l'attuale lato migliore per il suo inserimento nella nuvola. L'esecuzione prosegue nella Figura 14.16.

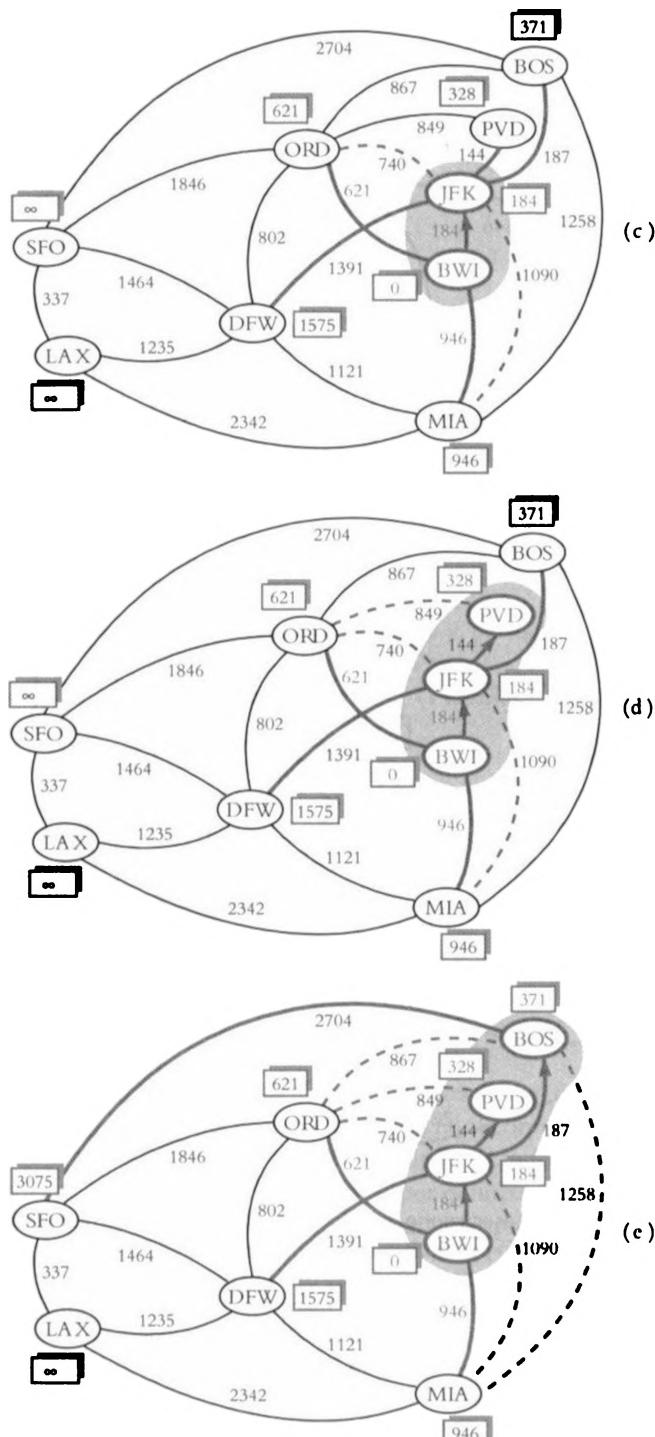
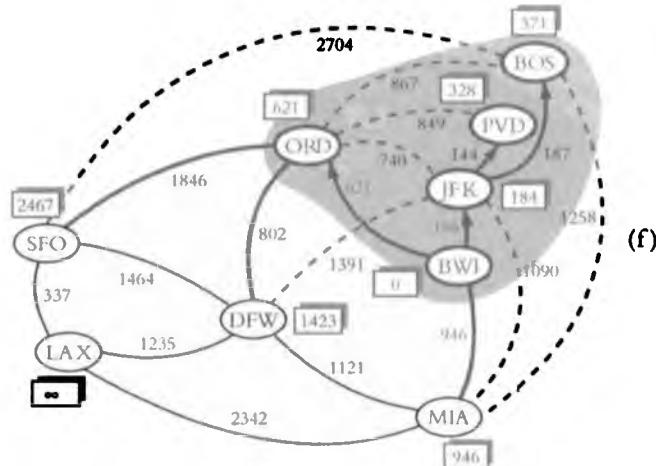
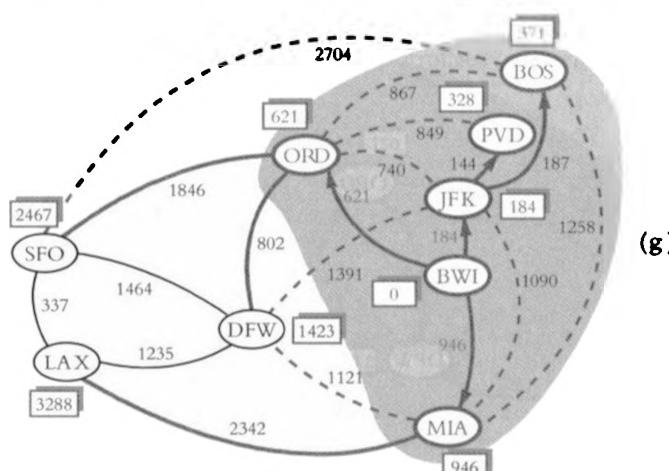


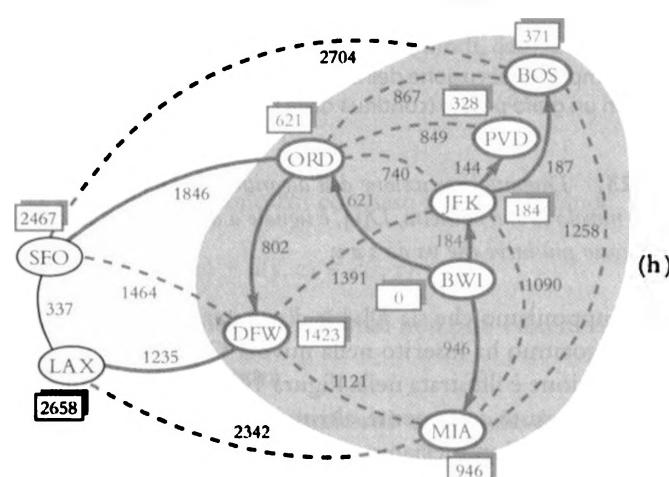
Figura 14.16 (prima parte): Un esempio di esecuzione dell'algoritmo di Dijkstra per la ricerca della lunghezza del percorso minimo in un grafo pesato (continua dalla Figura 14.15 e prosegue nella Figura 14.17).



(f)



(g)



(h)

Figura 14.16 (seconda parte): Un esempio di esecuzione dell'algoritmo di Dijkstra per la ricerca della lunghezza del percorso minimo in un grafo pesato (continua dalla Figura 14.15 e prosegue nella Figura 14.17).

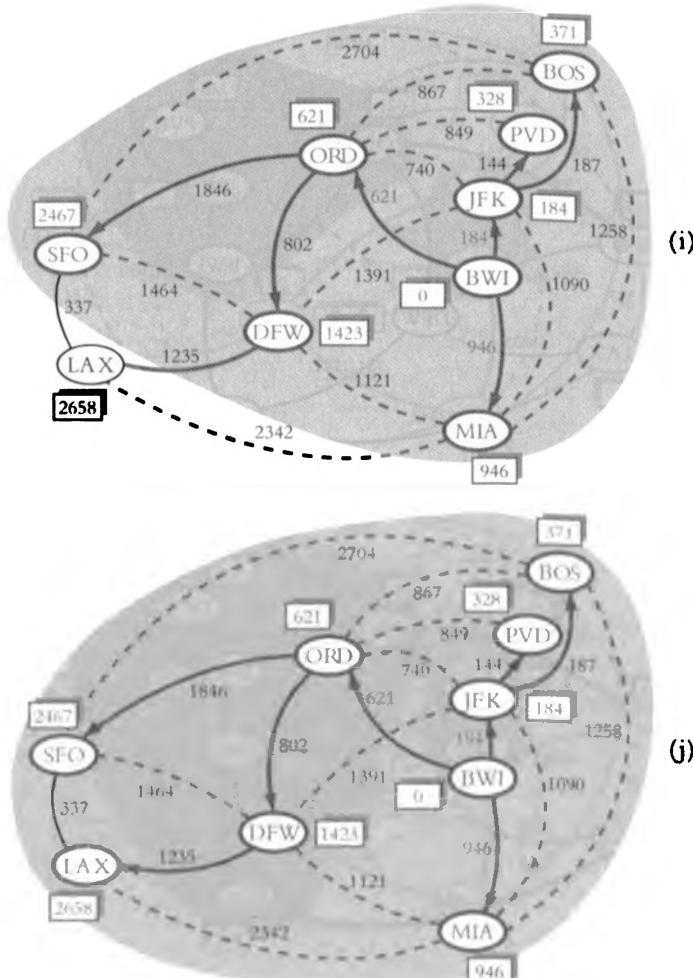


Figura 14.17: Un esempio di esecuzione dell'algoritmo di Dijkstra per la ricerca della lunghezza del percorso minimo in un grafo pesato (continua dalla Figura 14.16).

Proposizione 14.23: Durante l'esecuzione dell'algoritmo di Dijkstra, ogni volta che un vertice v viene inserito nella nuvola la sua etichetta, $D[v]$, è uguale a $d(s, v)$, che è la distanza di s da v , cioè la lunghezza del percorso più breve che va da s a v .

Dimostrazione: Supponiamo che sia $D[v] > d(s, v)$ per qualche vertice v in V , e sia z il primo vertice che l'algoritmo ha inserito nella nuvola C (dopo averlo estratto da Q) con $D[z] > d(s, z)$. La situazione è illustrata nella Figura 14.18 e indichiamo con P il più breve percorso che va da s a z : esiste certamente, altrimenti sarebbe $d(s, z) = \infty = D[z]$ e z non sarebbe stato estratto da Q . Consideriamo, quindi, il momento in cui z viene inserito in C e chiamiamo y il primo vertice di P (procedendo da s verso z) che, in tale momento, non appartiene a C . Infine, sia x il predecessore di y lungo il percorso P (osservando che potrebbe anche essere $x = s$): per come abbiamo definito y , sappiamo che x è interno a C .

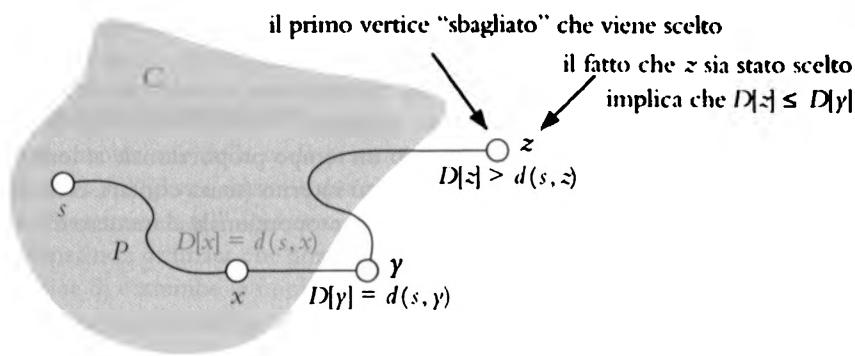


Figura 14.18: Un'illustrazione schematica della situazione utilizzata per dimostrare la Proposizione 14.23.

Inoltre, $D[x] = d(s, x)$, perché z è il *primo* vertice che entra nella nuvola in modo “scorretto”. Quando x è stato inserito in C , abbiamo controllato (ed eventualmente aggiornato) $D[y]$ in modo che, in quel momento, fosse:

$$D[y] \leq D[x] + w(x, y) = d(s, x) + w(x, y).$$

Però y è il vertice successivo lungo il più breve percorso che va da s a z , quindi:

$$D[y] = d(s, y).$$

Siamo ora giunti al momento in cui stiamo per scegliere z , e non y , come prossimo vertice da inserire in C . Quindi:

$$D[z] \leq D[y].$$

Dovrebbe essere evidente che un sotto-percorso di un percorso più breve deve essere un percorso più breve tra i suoi due vertici terminali. Quindi, dato che y si trova sul percorso più breve che va da s a z , si ha:

$$d(s, y) + d(y, z) = d(s, z).$$

Inoltre, $d(y, z) \geq 0$, perché non ci sono lati con peso negativo. Quindi:

$$D[z] \leq D[y] = d(s, y) \leq d(s, y) + d(y, z) = d(s, z).$$

Ma questo contraddice la definizione di z , quindi non può esistere alcun vertice z . ■

Tempo d'esecuzione dell'algoritmo di Dijkstra

In questo paragrafo analizzeremo la complessità temporale dell'algoritmo di Dijkstra. Indichiamo, rispettivamente, con n e m il numero di vertici e di lati del grafo G su cui operare e ipotizziamo che i pesi dei lati possano essere sommati e confrontati tra loro in un tempo costante. Visto l'alto livello di astrazione a cui abbiamo descritto l'algoritmo di Dijkstra nel Codice 14.12, l'analisi del suo tempo d'esecuzione richiede che si forniscano

maggiori dettagli sulla sua implementazione. Nello specifico, dobbiamo dire quali strutture dati vengono utilizzate e come sono implementate.

Per prima cosa ipotizziamo che il grafo sia rappresentato usando una struttura a lista di adiacenze o a mappa di adiacenze. Queste strutture ci consentono, durante la fase di rilassamento, di scandire i vertici adiacenti a u in un tempo proporzionale al loro numero, quindi il tempo speso per la *gestione* del ciclo `for` più interno (senza contare, cioè, il tempo necessario per eseguire ripetutamente il suo corpo) è proporzionale al numero di iterazioni che questo compie, numero dato da questa espressione:

$$\sum_{u \text{ in } V_G} \text{outdeg}(u).$$

che, per la Proposizione 14.9, è $O(m)$. Il ciclo `while` più esterno viene eseguito $O(n)$ volte, perché in ciascuna sua iterazione viene aggiunto alla nuvola un nuovo vertice. Quanto detto non esaurisce, però, i dettagli necessari per l'analisi dell'algoritmo, perché dobbiamo parlare dell'implementazione della struttura dati principale: la coda prioritaria Q .

Tornando a esaminare il Codice 14.12 in cerca delle operazioni eseguite sulla coda prioritaria, troviamo che gli n vertici vengono inizialmente tutti inseriti nella coda prioritaria: dato che questi sono gli unici inserimenti effettuati, la dimensione massima della coda è n . In ciascuna iterazione del ciclo `while` viene effettuata un'invocazione del metodo `removeMin`, che ha il compito di estrarre da Q il vertice u avente etichetta D minima. Poi, per ogni vertice v adiacente a u , eseguiamo un rilassamento del lato corrispondente e questo può portare a un aggiornamento della chiave associata a v all'interno della coda prioritaria, che, quindi, deve essere implementata come *coda prioritaria modificabile* (*adaptable priority queue*, vista nel Paragrafo 9.5): la chiave associata al vertice v può essere modificata usando il metodo `replaceKey(e , k)`, dove e è la voce associata al vertice v nella coda prioritaria. Nel caso peggiore ci potrà essere uno di tali aggiornamenti per ogni lato del grafo, quindi, complessivamente, il tempo d'esecuzione dell'algoritmo di Dijkstra è limitato superiormente dalla somma dei seguenti addendi:

- n inserimenti in Q
- n invocazioni del metodo `removeMin` di Q
- m invocazioni del metodo `replaceKey` di Q

Se Q è una coda prioritaria modificabile implementata con uno heap, ciascuna delle operazioni citate viene eseguita in un tempo $O(\log n)$, quindi il tempo totale richiesto per l'esecuzione dell'algoritmo di Dijkstra è $O((n + m) \log n)$, nel caso peggiore.

Consideriamo, ora, un'implementazione alternativa per la coda prioritaria modificabile Q , usando una sequenza non ordinata (si veda l'Esercizio P-9.52). Questo, ovviamente, rende $O(n)$ il tempo richiesto per estrarre l'elemento minimo, ma rende estremamente veloci gli aggiornamenti dei valori delle chiavi, a condizione che Q usi voci "consapevoli della propria posizione" (*location-aware entry*, illustrate nel Paragrafo 9.5.1). In particolare, è possibile implementare Q in modo che ciascuna modifica di chiave richiesta durante le fasi di rilassamento venga eseguita in un tempo $O(1)$: dopo aver individuato la voce all'interno di Q , si modifica semplicemente la sua chiave. Quindi, questa implementazione richiede,

per l'esecuzione dell'algoritmo di Dijkstra, un tempo complessivo $O(n^2 + m)$, che può essere semplificato in $O(n^2)$ perché il grafo G è semplice.

Confronto tra le due implementazioni

Per l'implementazione della coda prioritaria modificabile (con voci *location-aware*) usata dall'algoritmo di Dijkstra abbiamo due alternative: un'implementazione mediante heap, che consente l'esecuzione dell'algoritmo in un tempo $O((n + m) \log n)$, e un'implementazione con sequenza non ordinata, che porta l'algoritmo a un'esecuzione in un tempo $O(n^2)$. Dato che il codice di entrambe le implementazioni è abbastanza semplice da scrivere, sono due soluzioni sostanzialmente equivalenti dal punto di vista della difficoltà di programmazione e sono anche piuttosto simili in termini dei fattori costanti che influenzano i loro tempi d'esecuzione nel caso peggiore. Guardando soltanto al caso peggiore, preferiamo l'implementazione mediante heap quando il numero di lati del grafo è "piccolo" (cioè quando $m < n^2/\log n$), mentre preferiamo l'implementazione con una sequenza quando il numero di lati è "grande" (cioè quando $m > n^2/\log n$).

Proposizione 14.24: *Dato un grafo pesato G con n vertici e m lati (con pesi dei lati non negativi) e un vertice s di G , l'algoritmo di Dijkstra è in grado di calcolare la distanza di s da tutti gli altri vertici di G in un tempo limitato superiormente dalla migliore tra le due espressioni $O(n^2)$ e $O((n + m) \log n)$.*

Osserviamo anche che, usando un'implementazione più sofisticata di coda prioritaria, che prende il nome di *heap di Fibonacci*, si può realizzare l'algoritmo di Dijkstra in modo che il suo tempo d'esecuzione sia $O(m + n \log n)$.

Implementazione dell'algoritmo di Dijkstra in Java

Dopo aver presentato una descrizione dell'algoritmo di Dijkstra mediante pseudocodice, vediamone ora un'implementazione in Java, nell'ipotesi che venga passato come parametro un grafo i cui lati hanno come pesi dei numeri interi non negativi. La nostra implementazione dell'algoritmo, nel Codice 14.13, è un metodo, `shortestPathLengths`, che riceve come parametro un grafo e un vertice, che verrà usato come origine dei percorsi, e restituisce una mappa (`cloud`) che contiene la distanza $d(s, v)$ per ciascun vertice v che sia raggiungibile dall'origine s . Come coda prioritaria modificabile usiamo un esemplare della classe `Heap-AdaptablePriorityQueue`, che abbiamo sviluppato nel Paragrafo 9.5.2.

Come abbiamo fatto con gli algoritmi di questo capitolo, usiamo mappe implementate con tabelle hash per memorizzare informazioni ausiliarie: in questo caso, per mettere in relazione il vertice v con il limite superiore $D[v]$ della sua distanza da s e con la sua voce all'interno della coda prioritaria. Il tempo atteso $O(1)$ per l'esecuzione dell'accesso agli elementi di queste mappe può diventare un limite nel caso peggiore numerando i vertici da 0 a $n - 1$ e usandoli come indici in un array oppure memorizzando le informazioni all'interno dell'elemento contenuto in ciascun vertice.

Lo pseudocodice che definisce l'algoritmo di Dijkstra inizia con l'assegnazione $D[v] = \infty$ per tutti i vertici v diversi dall'origine s : usiamo, in questo caso, il valore speciale `Integer.MAX_VALUE`, previsto da Java, come valore numerico sufficientemente elevato da simulare l'infinito. Tuttavia, evitiamo di includere vertici con tale distanza "infinita" nella nuvola che viene restituita dal metodo come risultato prodotto dall'algoritmo. L'uso di questo valore

numerico limite si potrebbe evitare completamente evitando di aggiungere i vertici alla coda prioritaria fino a quando non viene sottoposto a rilassamento un vertice che li raggiunge (si veda l'Esercizio C-14.62).

Codice 14.13: Implementazione in Java dell'algoritmo di Dijkstra per il calcolo delle distanze di tutti i vertici da una sola origine. Ipotizziamo che il valore restituito da `e.getElement()` sia il peso corrispondente al lato `e`.

```

1  /** Calcola le distanze dal vertice src di tutti i vertici di g raggiungibili. */
2  public static <V> Map<Vertex<V>, Integer>
3      shortestPathLengths(Graph<V, Integer> g, Vertex<V> src) {
4      // d.get(v) è il limite superiore per la distanza di v da src
5      Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
6      // associa v al suo valore d
7      Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
8      // pq avrà i vertici come valori, con d.get(v) come chiave
9      AdaptablePriorityQueue<Integer, Vertex<V>> pq;
10     pq = new HeapAdaptablePriorityQueue<>();
11     // associa i vertici alla loro posizione in pq
12     Map<Vertex<V>, Entry<Integer, Vertex<V>> pqTokens;
13     pqTokens = new ProbeHashMap<>();
14
15     // per ogni vertice v del grafo, aggiunge una voce alla coda prioritaria, con
16     // l'origine src avente distanza 0 e tutti gli altri vertici distanza infinita
17     for (Vertex<V> v : g.vertices()) {
18         if (v == src)
19             d.put(v, 0);
20         else
21             d.put(v, Integer.MAX_VALUE);
22         pqTokens.put(v, pq.insert(d.get(v), v)); // memorizza la voce per poi aggiornarla
23     }
24     // ora inizia l'inserimento di vertici raggiungibili nella nuvola
25     while (!pq.isEmpty()) {
26         Entry<Integer, Vertex<V>> entry = pq.removeMin();
27         int key = entry.getKey();
28         Vertex<V> u = entry.getValue();
29         cloud.put(u, key); // questa è l'effettiva distanza di u da src
30         pqTokens.remove(u); // u non è più presente in pq
31         for (Edge<Integer> e : g.outgoingEdges(u)) {
32             Vertex<V> v = g.opposite(u, e);
33             if (cloud.get(v) == null) {
34                 // esegui il rilassamento per il lato (u, v)
35                 int wgt = e.getElement();
36                 if (d.get(u) + wgt < d.get(v)) { // percorso migliore per v ?
37                     d.put(v, d.get(u) + wgt); // aggiorna la distanza di v
38                     pq.replaceKey(pqTokens.get(v), d.get(v)); // aggiorna la voce di v in pq
39                 }
40             }
41         }
42     }
43     return cloud; // contiene soltanto i vertici raggiungibili
44 }
```

Ricostruzione dell'albero dei percorsi più brevi

La nostra descrizione dell'algoritmo di Dijkstra, presentata con pseudocodice nel Codice 14.12 e poi implementata in Java nel Codice 14.13, calcola, per ogni vertice v , il valore $D[v]$,

che è la lunghezza del percorso più breve che va dall'origine s al vertice v . Tuttavia, queste forme dell'algoritmo non costruiscono esplicitamente gli effettivi percorsi che consentono di ottenere tali distanze. Fortunatamente, è possibile rappresentare con una struttura dati compatta, che prende il nome di *albero dei percorsi più brevi* (*shortest-path tree*), l'insieme dei percorsi più brevi che, dall'origine s , raggiungono qualsiasi altro vertice raggiungibile del grafo. Questo è possibile perché, se uno dei percorsi più brevi che vanno da s a v passa per il vertice intermedio u , allora la sua parte iniziale deve essere necessariamente uno dei percorsi più brevi che vanno da s a u .

Ora dimostreremo che un albero dei percorsi più brevi avente radice nell'origine s può essere ricostruito in un tempo $O(n + m)$ a partire dai valori $D[v]$ che sono stati calcolati dall'algoritmo di Dijkstra usando s come origine. Come abbiamo fatto quando abbiamo rappresentato l'albero DFS e l'albero BFS, mettiamo in corrispondenza ciascun vertice $v \neq s$ con il genitore u (al limite, può essere $u = s$), tale che u sia il vertice che precede immediatamente v lungo uno dei percorsi più brevi che vanno da s a v . Se u è il vertice che precede immediatamente v lungo uno dei percorsi più brevi che vanno da s a v , allora deve essere:

$$D[u] + w(u, v) = D[v].$$

Viceversa, se l'equazione precedente è soddisfatta, allora il percorso più breve che va da s a v , seguito dal lato (u, v) , è uno dei percorsi più brevi che vanno da s a v .

Il metodo `spTree`, presentato nel Codice 14.14, è in grado di ricostruire un albero basato su questa logica, verificando tutti i lati *entranti* in ciascun vertice v e cercando un lato (u, v) che soddisfi quell'equazione. Il tempo d'esecuzione del metodo è $O(n + m)$, perché prendiamo in esame ciascun vertice e tutti i lati entranti in tali vertici (si veda la Proposizione 14.9).

Codice 14.14: Implementazione in Java dell'algoritmo che ricostruisce un albero dei percorsi più brevi a partire da una sola origine, sulla base delle distanze calcolate, ad esempio, dall'algoritmo di Dijkstra.

```

1  /**
2   * Ricostruisce un albero dei percorsi più brevi avente radice  $s$ , data la mappa  $d$ 
3   * con le distanze. L'albero è rappresentato come una mappa tra ciascun vertice  $v$ 
4   * raggiungibile da  $s$  (tranne  $s$ ) e il lato  $e = (u, v)$  usato per raggiungere  $v$  dal
5   * suo genitore  $u$  nell'albero. */
6 public static <V> Map<Vertex<V>, Edge<Integer>>
7     spTree(Graph<V, Integer> g, Vertex<V> s, Map<Vertex<V>, Integer> d) {
8     Map<Vertex<V>, Edge<Integer>> tree = new ProbableHashMap<>();
9     for (Vertex<V> v : d.keySet())
10        if (v != s)
11            for (Edge<Integer> e : g.incomingEdges(v)) { // solo i lati ENTRANTI
12                Vertex<V> u = g.opposite(v, e);
13                int wgt = e.getElement();
14                if (d.get(v) == d.get(u) + wgt)
15                    tree.put(v, e); // il lato  $e$  è stato usato per raggiungere  $v$ 
16            }
17     return tree;
18 }
```

14.7 Alberi ricoprenti minimi

Immaginiamo di voler collegare tutti i computer di un nuovo edificio per uffici usando la minima quantità di cavo. Possiamo usare, come modello di questo problema, un grafo G non orientato e pesato, i cui vertici rappresentino i computer e i cui lati rappresentino tutte le possibili coppie di computer (u, v) , con il peso $w(u, v)$ del lato (u, v) uguale alla quantità di cavo necessaria a collegare il computer u al computer v . Non siamo particolarmente interessati al calcolo dell'albero dei percorsi più brevi per qualche specifico vertice v , quanto, invece, a individuare un albero T che contenga tutti i vertici di G e una selezione dei suoi lati che ne renda minimo il peso totale, tra tutti gli alberi possibili. Gli algoritmi che consentono di risolvere questo problema saranno l'argomento di questo capitolo.

Definizione del problema

Dato un grafo G non orientato e pesato, siamo interessati a individuare un albero T che contenga tutti i vertici di G e minimizzi la somma

$$w(T) = \sum_{(u,v) \text{ in } T} w(u, v).$$

Un albero come questo, che contiene tutti i vertici di un grafo connesso G , è detto **albero ricoprente** (*spanning tree*) e il problema di individuare un albero ricoprente T avente peso totale minimo è noto come **problema MST** (*minimum spanning tree*) o “problema del minimo albero ricoprente”.

Lo sviluppo di algoritmi efficienti per risolvere il problema del minimo albero ricoprente è precedente al concetto stesso di informatica moderna. In questo paragrafo vedremo due algoritmi classici che risolvono il problema MST: entrambi sono applicazioni del **metodo greedy**, che, come già detto sinteticamente nel paragrafo precedente, è basato sulla scelta ripetuta di oggetti da aggiungere a un contenitore che aumenta di dimensione, selezionando di volta in volta quello che minimizza una determinata funzione di costo. Il primo algoritmo di cui parliamo è l'algoritmo di Prim-Jarník, che fa crescere un albero ricoprente minimo a partire da un singolo vertice, usato come radice dell'albero, usando una strategia simile a quella dell'algoritmo di Dijkstra. Il secondo algoritmo che presenteremo è l'algoritmo di Kruskal, che fa “crescere” l'albero in più “zone” (dette *cluster*), prendendo in esame i lati in ordine di peso non decrescente.

Per semplificare la descrizione di questi algoritmi, ipotizziamo, in questo paragrafo, che il grafo G da elaborare sia non orientato (cioè che tutti i suoi lati siano non orientati) e semplice (cioè che non contenga auto-anelli né lati paralleli). Quindi, possiamo rappresentare i lati di G come coppie non ordinate di vertici, (u, v) .

Prima di affrontare i dettagli di questi due algoritmi, però, discutiamo una proprietà fondamentale degli alberi ricoprenti minimi, su cui si basano entrambi.

Una proprietà importante degli alberi ricoprenti minimi

I due algoritmi che discuteremo e che hanno l'obiettivo di individuare un MST sono basati sul metodo greedy, che, in questo caso, dipende dalla proprietà seguente, illustrata nella Figura 14.19.

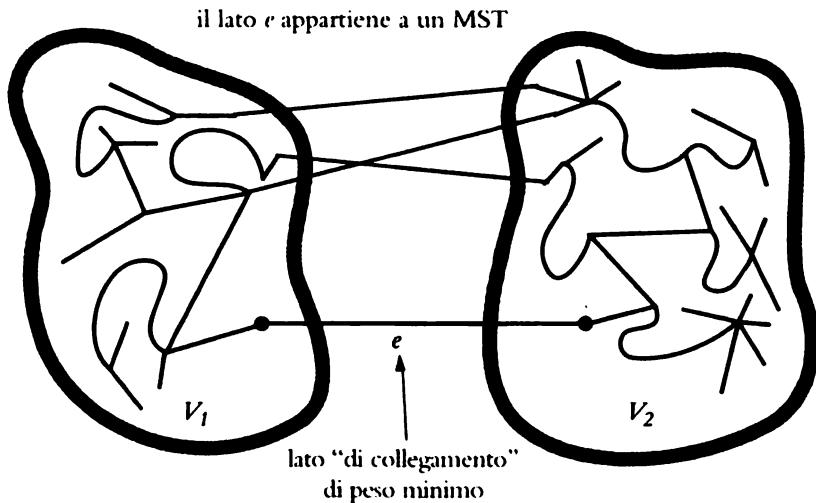


Figura 14.19: Un'illustrazione della proprietà fondamentale degli alberi ricoprenti minimi (MST).

Proposizione 14.25: Sia G un grafo pesato e connesso e sia V_1 e V_2 una partizione che suddivide i vertici di G in due insiemi non vuoti e disgiunti. Inoltre, sia e un lato di G avente peso minimo tra quelli che hanno un vertice terminale in V_1 e l'altro in V_2 . Esiste un albero ricoprente minimo, T , che ha e come uno dei suoi lati.

Dimostrazione: Sia T un albero ricoprente minimo di G . Se T non contiene il lato e , l'aggiunta di e a T deve creare un ciclo, quindi esiste qualche lato $f \neq e$ di tale ciclo che ha un vertice terminale in V_1 e l'altro in V_2 . Inoltre, per la scelta di e , si ha che $w(e) \leq w(f)$. Se eliminiamo f da $T \cup \{e\}$, otteniamo un albero ricoprente il cui peso totale non è maggiore di quello precedente. Dato che T era un albero ricoprente minimo, anche questo nuovo albero deve essere un albero ricoprente minimo. ■

In effetti, se i pesi dei lati di G sono tutti diversi, allora l'albero ricoprente minimo è unico: lasciamo all'Esercizio C-14.64 la dimostrazione di questa seconda proprietà, meno critica. Inoltre, osserviamo che la Proposizione 14.25 rimane valida anche se il grafo G contiene lati o cicli con peso negativo, diversamente da quanto visto per l'algoritmo di Dijkstra.

14.7.1 L'algoritmo di Prim-Jarník

Durante l'esecuzione dell'algoritmo di Prim-Jarník facciamo crescere un albero ricoprente minimo a partire da un albero elementare contenente un vertice "radice" s , qualsiasi. L'idea fondamentale è analoga a quella che governa il funzionamento dell'algoritmo di Dijkstra. Cominciamo con un vertice s , definendo la "nuvola" iniziale di vertici C . Poi, durante ciascuna iterazione dell'algoritmo, scegliamo un lato di peso minimo $e = (u, v)$, che connette un vertice u appartenente alla nuvola C a un vertice v esterno a C . Il vertice v viene portato all'interno della nuvola e il processo si ripete fino alla costruzione dell'intero albero.

ricoprente, cioè fino ad aver inserito tutti i vertici nella nuvola C . Di nuovo, entra in gioco la proprietà fondamentale degli alberi ricoprenti minimi, che abbiamo messo in evidenza, perché, scegliendo sempre il lato di peso minimo che congiunge un vertice interno a C con un vertice esterno a C , siamo sicuri di aggiungere sempre all'albero un lato valido per preservare la sua proprietà.

Per implementare in modo efficiente questo approccio, possiamo prendere di nuovo spunto dall'algoritmo di Dijkstra. Usiamo un'etichetta $D[v]$ per ogni vertice v esterno alla nuvola C , in modo che $D[v]$ contenga sempre il peso del minimo lato osservato fino a quel momento tra quelli che collegano v alla nuvola C (mentre, ricordiamo, nell'algoritmo di Dijkstra l'etichetta conteneva la lunghezza del percorso completo di costo minimo dal vertice s a v , tra quelli contenenti il lato (u, v)). Queste etichette servono come chiavi in una coda prioritaria usata per decidere quale vertice sarà il successivo a entrare nella nuvola. Il Codice 14.15 riporta lo pseudocodice che definisce l'algoritmo di Prim-Jarník.

Codice 14.15: L'algoritmo di Prim-Jarník per risolvere il problema dell'individuazione di un albero ricoprente minimo (MST).

Algoritmo `PrimJarnik(G)`:

Input: Un grafo G ,连通的, non orientato e pesato, con n vertici e m lati

Output: Un albero ricoprente minimo T per G

scegli un vertice s di G

$D[s] = 0$

for ogni vertice $v \neq s$ do

$D[v] = \infty$

inizializza $T = \emptyset$

crea una coda prioritaria Q con una voce $(D[v], v)$ per ogni vertice v

per ogni vertice v , gestisci l'informazione `connect(v)`: il lato che genera $D[v]$ (se esiste)
while Q non è vuota do

u = valore contenuto nella voce restituita da `Q.removeMin()`

connetti il vertice u a T usando il lato `connect(u)`

for ogni lato $e' = (u, v)$ tale che v appartenga a Q do

{ controlla se il lato (u, v) connette v a T in modo migliore }

if $w(u, v) < D[v]$ then

$D[v] = w(u, v)$

`connect(v) = e'`

cambia la chiave associata a v in Q : diventa $D[v]$

return l'albero T

Analisi dell'algoritmo di Prim-Jarník

I problemi da risolvere per l'implementazione dell'algoritmo di Prim-Jarník sono simili a quelli già analizzati per l'algoritmo di Dijkstra e riguardano principalmente la coda prioritaria modificabile Q (introdotta nel Paragrafo 9.5.1). Per prima cosa l'algoritmo effettua n inserimenti in Q , poi effettua complessivamente n operazioni di estrazione del minimo e può effettuare al massimo m modifiche di chiave: questi passi danno il contributo princi-

pale al tempo d'esecuzione totale dell'algoritmo. Con una coda prioritaria basata su heap, ogni operazione viene eseguita in un tempo $O(\log n)$, quindi il tempo totale è $O((n + m) \log n)$, che è $O(m \log n)$ per un grafo connesso. In alternativa, si può ottenere un tempo d'esecuzione $O(n^2)$ usando, come coda prioritaria, una lista non ordinata.

Visualizzazione del funzionamento dell'algoritmo di Prim-Jarník

Le Figure 14.20 e 14.21 illustrano un esempio di funzionamento dell'algoritmo di Prim-Jarník.

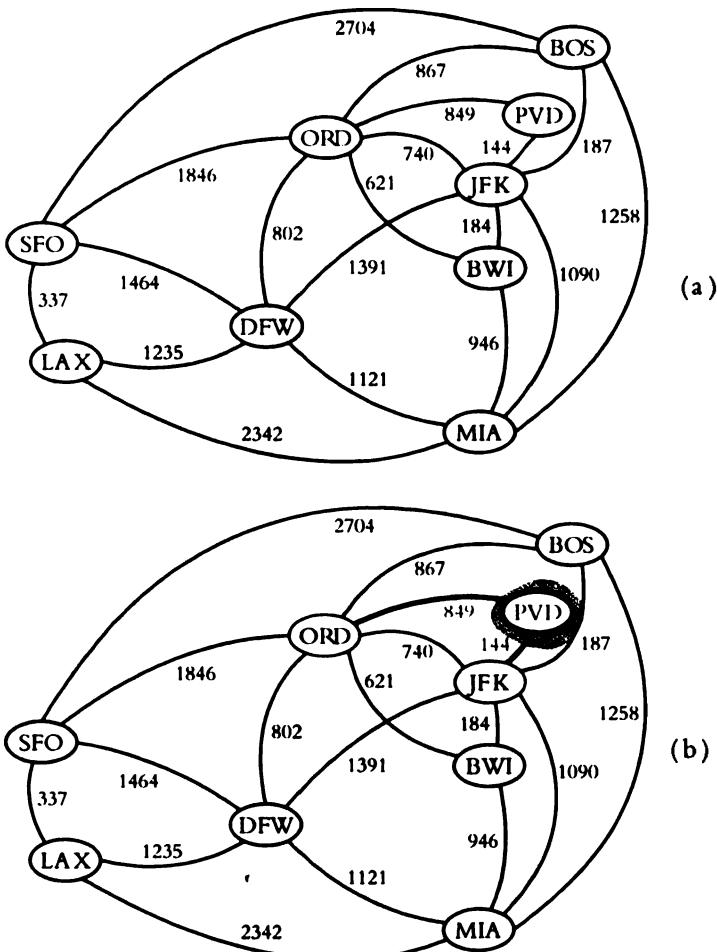


Figura 14.20 (prima parte): Visualizzazione dell'esecuzione dell'algoritmo di Prim-Jarník per l'individuazione di un MST, a partire dal vertice PVD (prosegue nella Figura 14.21).

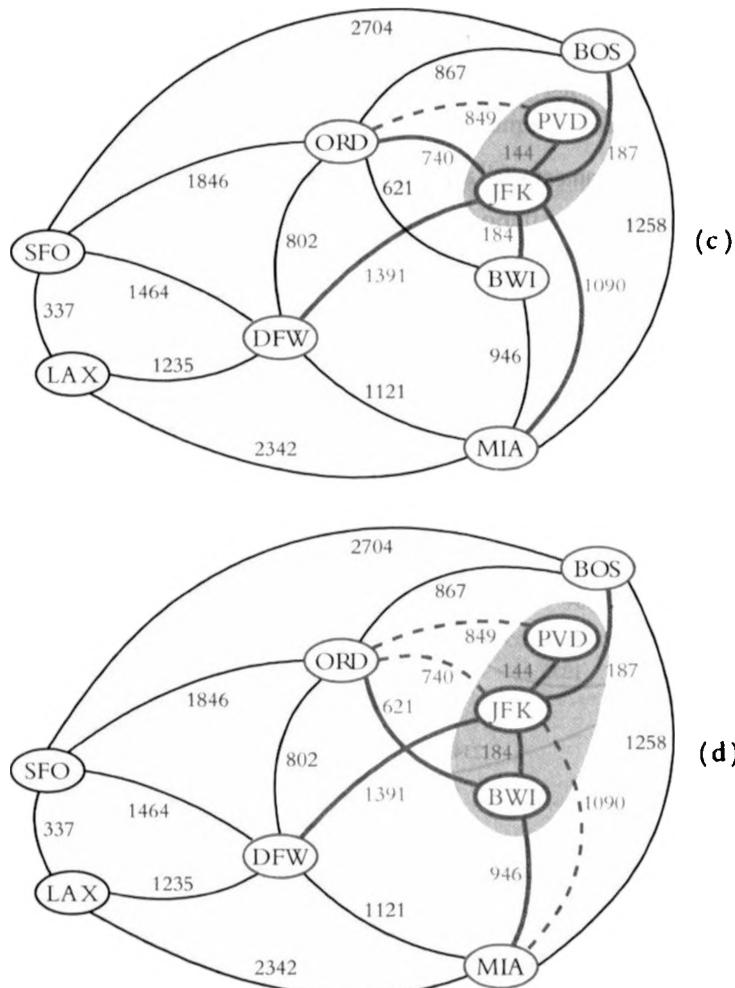


Figura 14.20 (seconda parte): Visualizzazione dell'esecuzione dell'algoritmo di Prim-Jarník per l'individuazione di un MST, a partire dal vertice PVD (prosegue nella Figura 14.21).

14.7.2 L'algoritmo di Kruskal

In questo paragrafo presenteremo l'**algoritmo di Kruskal** per costruire un albero ricoprente minimo (*minimum spanning tree*, MST). Mentre l'algoritmo di Prim-Jarník costruisce un MST facendo crescere un unico albero iniziale finché non arriva a ricoprire l'intero insieme di vertici del grafo, l'algoritmo di Kruskal gestisce molti piccoli alberi (detti *cluster*, cioè gruppi) in una *foresta*, fondendo ripetutamente coppie di alberi fino alla costruzione di un unico albero che ricopre l'intero grafo.

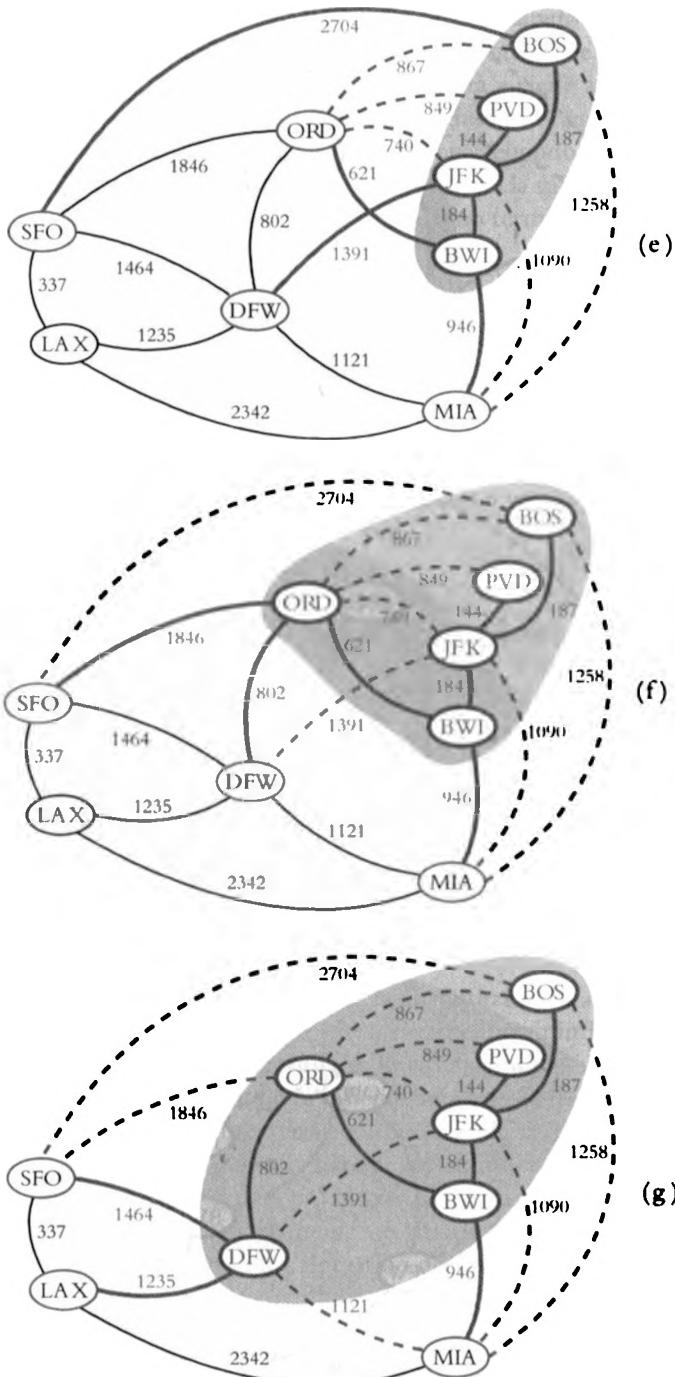


Figura 14.21 (prima parte): Visualizzazione dell'esecuzione dell'algoritmo di Prim-Jarník per l'individuazione di un MST, a partire dal vertice PVD (continua dalla Figura 14.20).

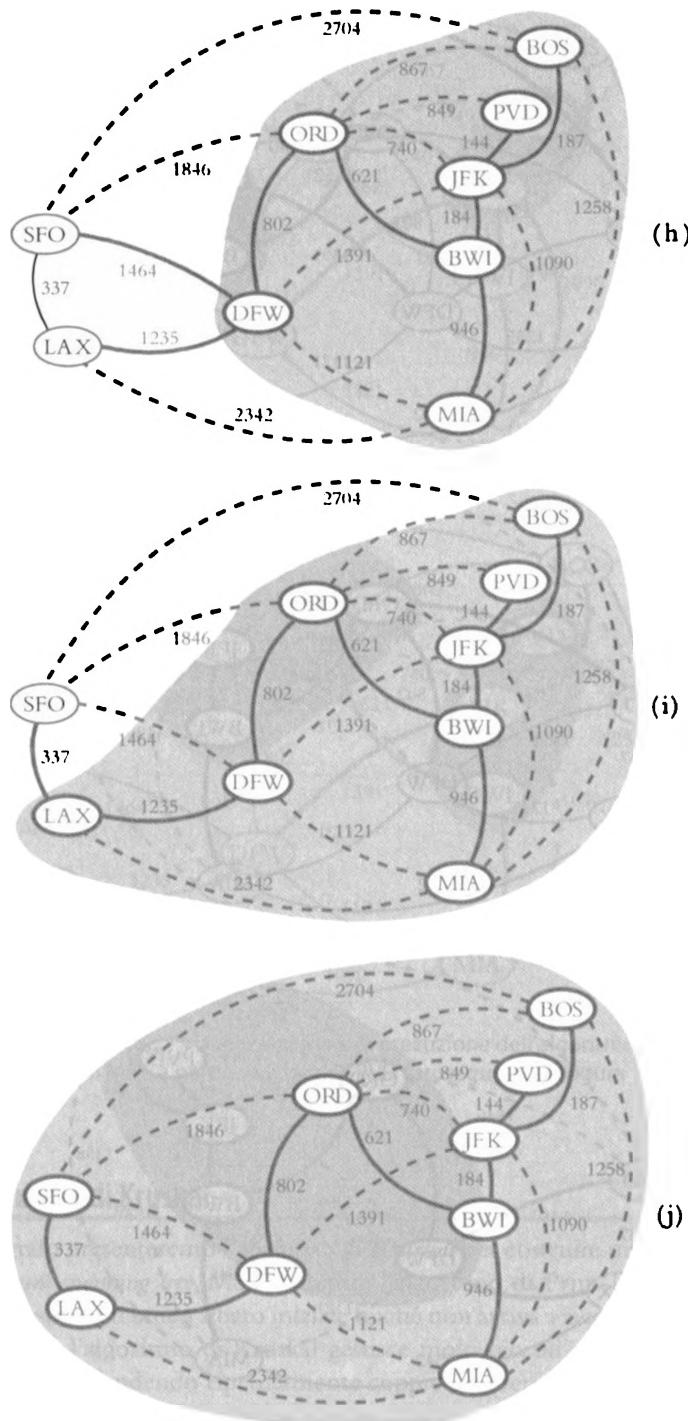


Figura 14.21 (seconda parte): Visualizzazione dell'esecuzione dell'algoritmo di Prim-Jarník per l'individuazione di un MST, a partire dal vertice PVD (continua dalla Figura 14.20).

Inizialmente, ciascun vertice costituisce il proprio piccolo albero (o *cluster*). L'algoritmo, poi, prende in esame ogni lato, a turno, in ordine di peso crescente. Se un lato *e* connette due vertici che appartengono a due cluster diversi, allora *e* viene aggiunto all'insieme di lati che costituirà l'albero ricoprente minimo e i due cluster in oggetto vengono fusi, con l'aggiunta di *e*. Se, invece, *e* connette due vertici che appartengono allo stesso cluster, allora viene ignorato. Dopo aver aggiunto all'albero che costituirà la soluzione un numero di lati sufficiente a dar luogo a un albero ricoprente, l'algoritmo termina e restituisce tale albero, che risulta essere un MST.

Il Codice 14.16 definisce l'algoritmo di Kruskal mediante pseudocodice, mentre un esempio di esecuzione è illustrato nelle Figure 14.22, 14.23 e 14.24.

Codice 14.16: L'algoritmo di Kruskal per risolvere il problema dell'individuazione di un albero ricoprente minimo (MST).

Algoritmo Kruskal(G):

Input: Un grafo G , semplice, connesso e pesato, con n vertici e m lati

Output: Un albero ricoprente minimo T per G

for ogni vertice v in G do

 definisci un cluster elementare $C(v) = \{v\}$

 crea una coda prioritaria Q contenente tutti i lati di G , usando i pesi come chiavi

 inizializza $T = \emptyset$ { alla fine T conterrà i lati di un MST }

 while T ha meno di $n - 1$ lati do

(u, v) = valore restituito da $Q.\text{removeMin}()$

$C(u)$ = cluster contenente u

$C(v)$ = cluster contenente v

 if $C(u) \neq C(v)$ then

 aggiungi il lato (u, v) a T

 fondi $C(u)$ e $C(v)$ in un unico cluster insieme al lato (u, v)

 return l'albero T

Come nel caso dell'algoritmo di Prim-Jarník, la correttezza dell'algoritmo di Kruskal è basata sulla proprietà fondamentale degli alberi ricoprenti minimi, enunciata nella Proposizione 14.25. Ogni volta che l'algoritmo di Kruskal aggiunge un lato, (u, v) , all'albero T , si può definire un partizionamento dell'insieme dei vertici V (come nella proposizione citata), in modo che V_1 sia il cluster che contiene v e V_2 sia l'insieme che contiene il resto dei vertici presenti in V . Questo chiaramente definisce un partizionamento di V mediante due insiemi disgiunti e, cosa più importante, dato che estraiamo i lati da Q in ordine di peso crescente, *e* è necessariamente un lato di peso minimo avente un vertice terminale in V_1 e l'altro in V_2 . Quindi, l'algoritmo di Kruskal aggiunge sempre all'albero che sta costruendo un lato valido.

Tempo d'esecuzione dell'algoritmo di Kruskal

Due sono le componenti principali che contribuiscono al tempo d'esecuzione dell'algoritmo di Kruskal. La prima è la necessità di prendere in esame i lati in ordine di peso non decrescente e la seconda è la gestione del partizionamento in cluster. L'analisi del tempo d'esecuzione, però, richiede di fornire maggiori dettagli sull'implementazione.

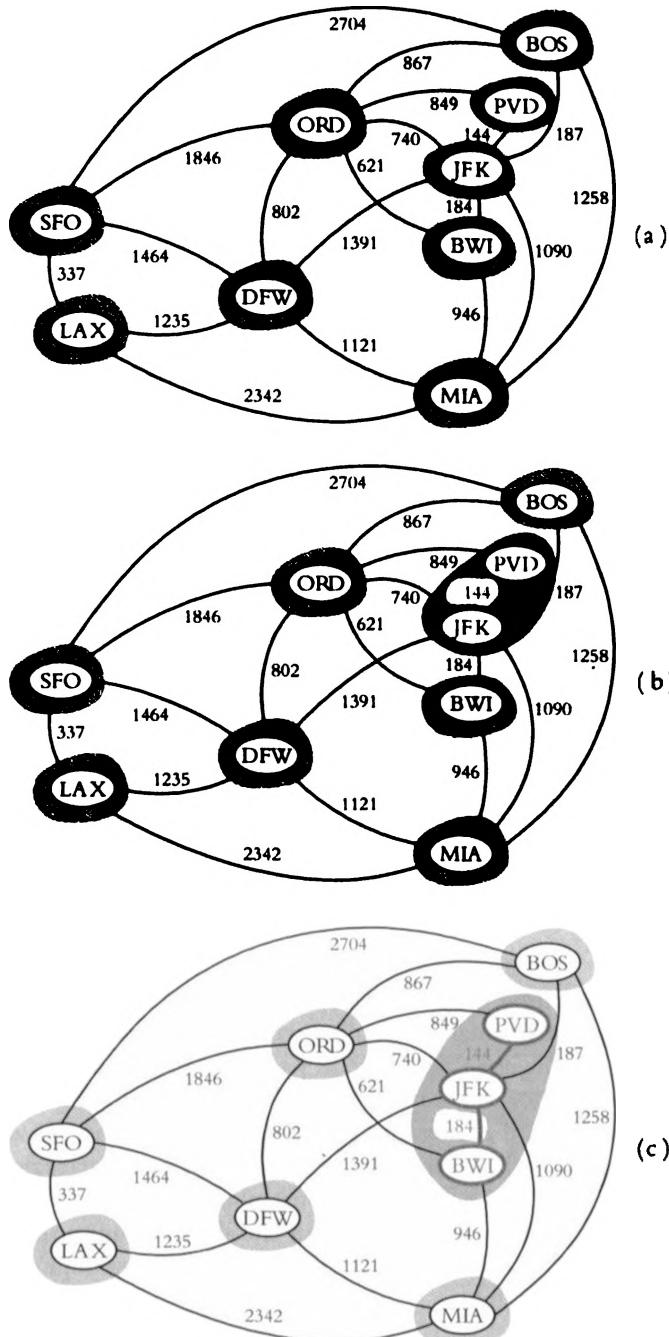


Figura 14.22 (prima parte): Visualizzazione (che prosegue nella Figura 14.23) dell'esecuzione dell'algoritmo di Kruskal per l'individuazione di un MST in un grafo avente numeri interi come pesi. I cluster sono le regioni di colore grigio e i lati del grafo di spessore maggiore sono quelli presi in esame nell'iterazione che è oggetto della singola figura.

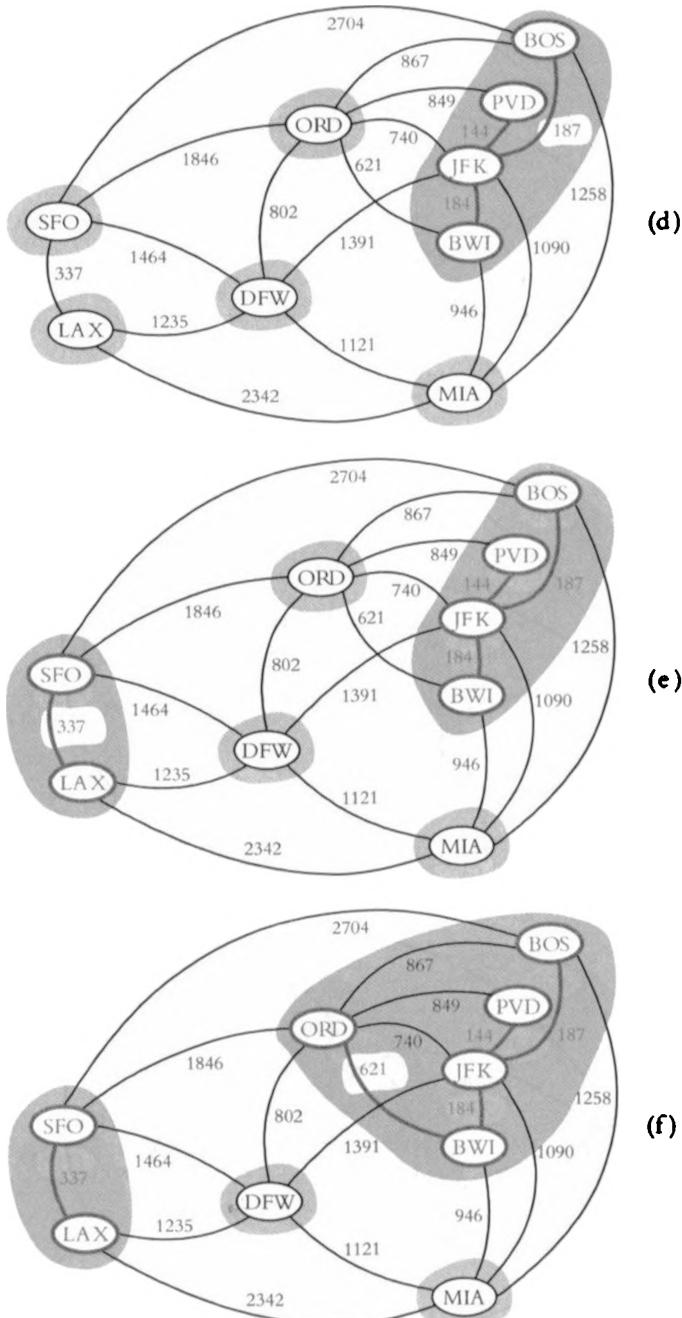


Figura 14.22 (seconda parte): Visualizzazione (che prosegue nella Figura 14.23) dell'esecuzione dell'algoritmo di Kruskal per l'individuazione di un MST in un grafo avente numeri interi come pesi. I cluster sono le regioni di colore grigio e i lati del grafo di spessore maggiore sono quelli presi in esame nell'iterazione che è oggetto della singola figura.

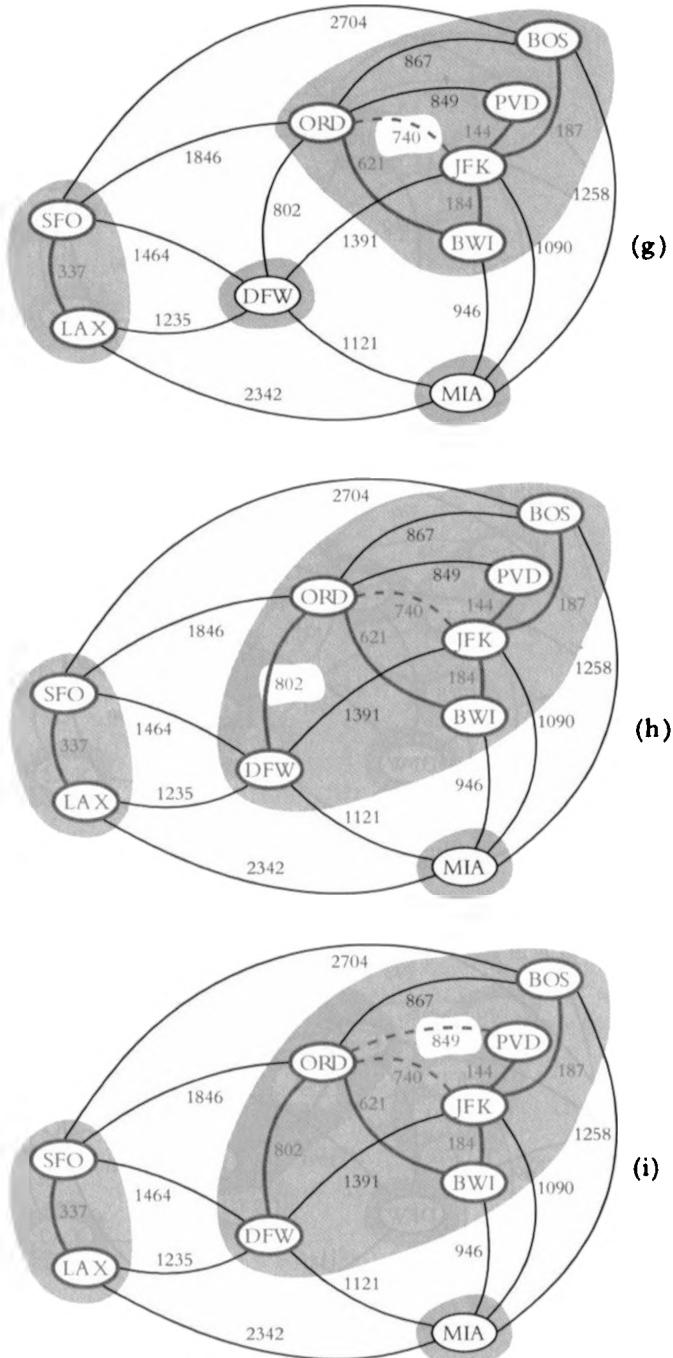


Figura 14.23 (prima parte): Visualizzazione dell'esecuzione dell'algoritmo di Kruskal per l'individuazione di un MST. I lati tratteggiati sono quelli che sono stati presi in esame e rifiutati dall'algoritmo (continua dalla Figura 14.22 e prosegue nella Figura 14.24).

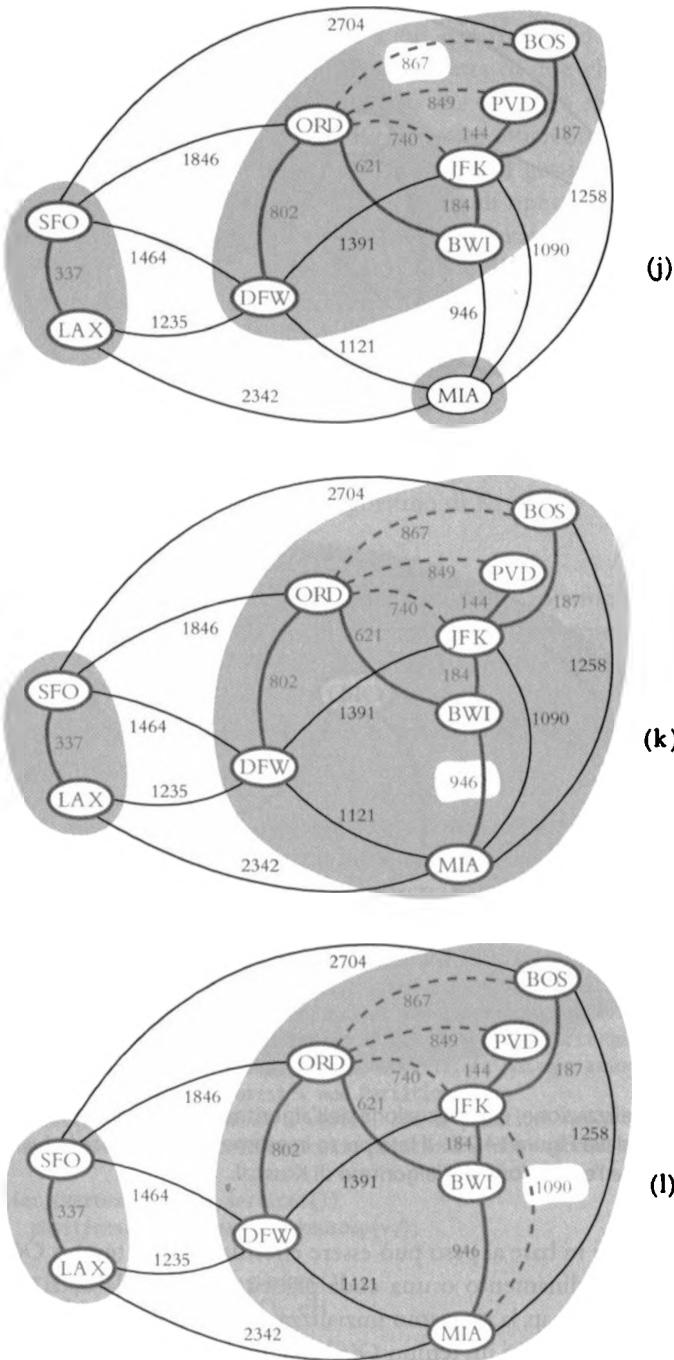


Figura 14.23 (seconda parte): Visualizzazione dell'esecuzione dell'algoritmo di Kruskal per l'individuazione di un MST. I lati tratteggiati sono quelli che sono stati presi in esame e rifiutati dall'algoritmo (continua dalla Figura 14.22 e prosegue nella Figura 14.24).

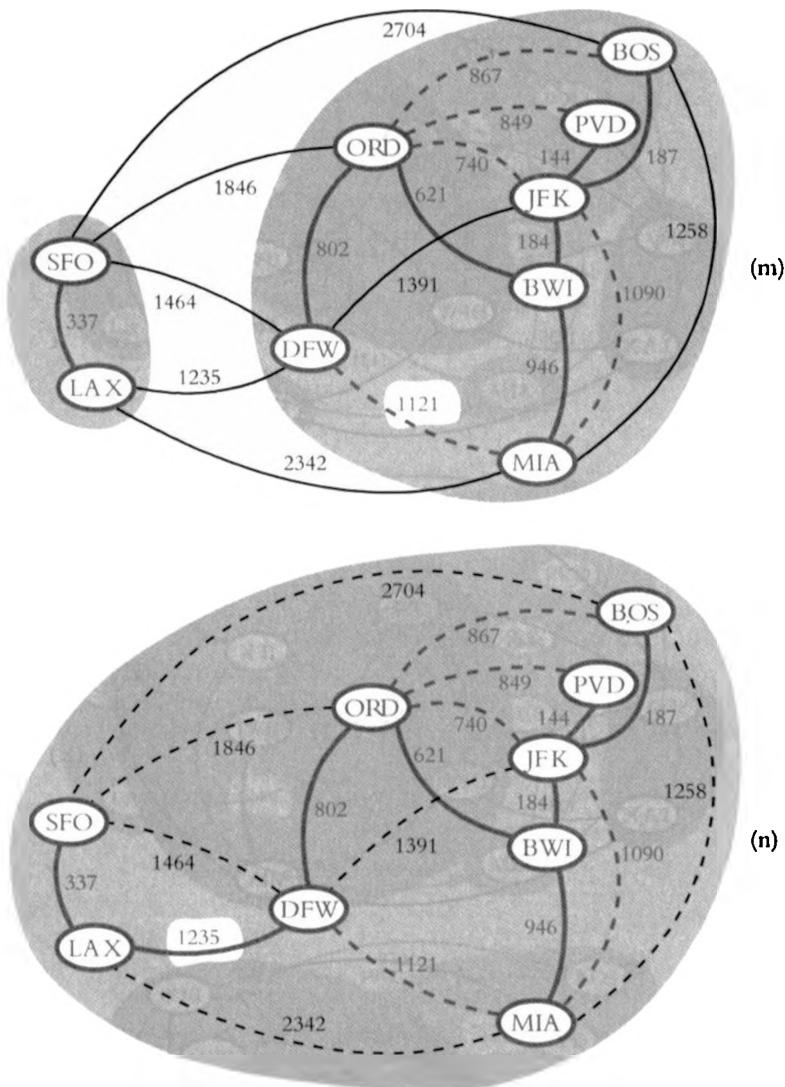


Figura 14.24: Visualizzazione dell'esecuzione dell'algoritmo di Kruskal per l'individuazione di un MST (continua dalla Figura 14.23). Il lato preso in esame in (n) provoca la fusione degli ultimi due cluster e conclude l'esecuzione dell'algoritmo di Kruskal.

L'ordinamento dei lati in base al peso può essere effettuato in un tempo $O(m \log m)$, usando un algoritmo di ordinamento o una coda prioritaria Q . Se la coda prioritaria viene implementata con uno heap, la possiamo inizializzare in un tempo $O(m \log m)$ effettuando ripetuti inserimenti, oppure in un tempo $O(m)$ usando la costruzione dello heap *bottom-up* (descritta nel Paragrafo 9.3.4); ciascuna delle successive ripetute invocazioni di `removeMin` richiede un tempo $O(\log m)$, perché la coda prioritaria ha dimensione $O(m)$. Osserviamo che in un grafo semplice, dal momento che m è una quantità $O(n^2)$, scrivere $O(\log m)$ è sempre equivalente a scrivere $O(\log n)$, quindi il tempo d'esecuzione dovuto agli accessi ordinati ai lati è $O(m \log n)$.

Ciò che rimane è la gestione dei cluster. Per implementare l'algoritmo di Kruskal dobbiamo essere in grado di trovare i cluster a cui appartengono i vertici u e v , vertici terminali di un lato e , in modo da verificare se si tratta di due cluster distinti e, in tal caso, fonderli insieme. Nessuna delle strutture dati che abbiamo studiato finora è particolarmente adatta a risolvere questo problema in modo efficiente, tuttavia concluderemo questo capitolo formalizzando proprio il problema di gestire *partizioni disgiunte* e introducendo strutture dati efficienti per l'esecuzione di operazioni di tipo *union-find*, cioè "cerca e unisci". Nel contesto dell'algoritmo di Kruskal, eseguiamo al massimo $2m$ operazioni "find" (cioè di ricerca) e $n - 1$ operazioni "union", cioè di fusione. Vedremo che una semplice struttura dedicata all'esecuzione di operazioni di tipo *union-find* è in grado di eseguire una combinazione di tali due operazioni in un tempo $O(m + n \log n)$, come enunciato nella Proposizione 14.26, e una struttura più avanzata può farlo anche più velocemente.

In un grafo连通的, è sempre $m \geq n - 1$, quindi il limite temporale $O(m \log n)$ dovuto all'ordinamento dei lati domina il tempo richiesto dalla gestione dei cluster. Possiamo, infine, concludere che il tempo d'esecuzione dell'algoritmo di Kruskal è $O(m \log n)$.

Implementazione in Java

Il Codice 14.17 presente un'implementazione in Java dell'algoritmo di Kruskal. L'albero ricoprente minimo viene restituito come lista di lati. Come conseguenza delle modalità di funzionamento dell'algoritmo di Kruskal, i lati sono elencati in tale lista in ordine di peso non decrescente.

La nostra implementazione usa una classe, `Partition`, per gestire la partizione di cluster: tale classe sarà presentata nel Paragrafo 14.7.3.

Codice 14.17: Implementazione in Java dell'algoritmo di Kruskal per l'individuazione di un albero ricoprente minimo. La classe `Partition` è discussa nel Paragrafo 14.7.3.

```

1  /** Calcola un MST di un grafo g usando l'algoritmo di Kruskal. */
2  public static <V> Positionallist<Edge<Integer>> MST(Graph<V, Integer> g) {
3      // l'albero tree è quello in cui costruiamo il risultato, passo dopo passo
4      Positionallist<Edge<Integer>> tree = new LinkedPositionallist<>();
5      // le voci di pq sono lati del grafo, con i pesi come chiavi
6      PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
7      // foresta di componenti del grafo, ottimizzata per operazioni union-find
8      Partition<Vertex<V>> forest = new Partition<>();
9      // mette in corrispondenza i vertici con le posizioni nella foresta
10     Map<Vertex<V>, Position<Vertex<V>>> positions = new ProbeHashMap<>();
11
12    for (Vertex<V> v : g.vertices())
13        positions.put(v, forest.makeGroup(v));
14
15    for (Edge<Integer> e : g.edges())
16        pq.insert(e.getElement(), e);
17
18    int size = g.numVertices();
19    // finché l'albero non è ricoprente e rimangono lati da esaminare...
20    while (tree.size() != size - 1 && !pq.isEmpty()) {
21        Entry<Integer, Edge<Integer>> entry = pq.removeMin();
22        Edge<Integer> edge = entry.getValue();
23        Vertex<V>[] endpoints = g.endVertices(edge);

```

```

    Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
    Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
    if (a != b) {
        tree.addLast(edge);
        forest.union(a, b);
    }
}

return tree;
}

```

14.7.3 Partizioni disgiunte e strutture *union-find*

In questo paragrafo consideriamo una struttura dati adatta alla gestione di una *partizione* di elementi all'interno di una raccolta di insiemi disgiunti. La motivazione che ci spinge a farlo è l'implementazione dell'algoritmo di Kruskal per la costruzione di un albero ricoprente minimo: tale algoritmo deve gestire una foresta di alberi disgiunti, con la necessità di effettuare fusioni di alberi adiacenti. Più in generale, il problema del partizionamento disgiunto si può applicare a vari casi di crescita di strutture discrete.

Formalizziamo il problema usando il modello seguente. Una struttura dati che rappresenta una partizione gestisce un universo di elementi organizzati in insiemi disgiunti (cioè un elemento appartiene a uno e soltanto uno di questi insiemi). Diversamente dal tipo di dato astratto "insieme", non ci aspettiamo di essere in grado di scandire l'intero contenuto di un insieme, né di verificare in modo efficiente se un dato insieme contenga un determinato elemento. Per evitare confusione con quel concetto di insieme, chiameremo *cluster* gli insiemi di questa nostra partizione. Non ci serve, però, una struttura esplicita per ciascun cluster, ci basta, invece, che l'organizzazione dei cluster sia implicita. Per distinguere tra un cluster e l'altro, facciamo l'ipotesi che, in qualsiasi istante, ogni cluster sia associato a un elemento designato a rappresentarlo, che chiamiamo *leader* del cluster.

Dal punto di vista formale, definiamo i metodi dell'ADT *partizione* (*partition*) usando posizioni, ciascuna delle quali contiene un elemento, x . Il tipo di dato astratto "partizione" mette a disposizione i seguenti metodi:

- makeCluster(x):** Crea un nuovo cluster contenente soltanto l'elemento x e restituisce la sua posizione.
- union(p, q):** Fonde insieme i cluster contenenti le posizioni p e q .
- find(p):** Restituisce la posizione del *leader* del cluster che contiene la posizione p .

Implementazione mediante sequenze

Una semplice implementazione di una partizione che abbia complessivamente n elementi usa una collezione di sequenze, una per ciascun cluster, con la sequenza associata al cluster A che memorizza le posizioni degli elementi di A . Come si può vedere nella Figura 14.25, ogni oggetto p di tipo "posizione" memorizza al proprio interno un riferimento all'elemento associato x e un riferimento alla sequenza che contiene p , perché tale sequenza rappresenta il cluster che contiene l'elemento di p .

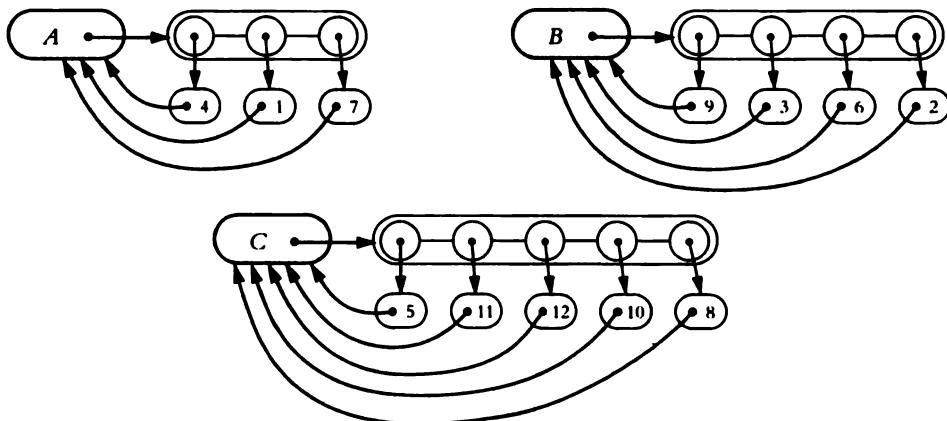


Figura 14.25: Implementazione mediante sequenze di una partizione costituita da tre cluster: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$ e $C = \{5, 8, 10, 11, 12\}$.

Con questa rappresentazione è facile implementare le operazioni `makeCluster(x)` e `find(p)` in modo che vengano eseguite in un tempo $O(1)$, facendo in modo che la prima posizione di una sequenza ricopri il ruolo di *leader*. L'operazione `union(p, q)` deve unire due sequenze in una sola e aggiornare i riferimenti *cluster* al cluster di appartenenza nelle posizioni di uno dei due cluster: sceglieremo di implementare questa operazione eliminando tutte le posizioni dalla sequenza di dimensione minore, inserendole in quella di dimensione maggiore. Ogni volta che togliamo una posizione dal cluster più piccolo, A , e la inseriamo nel cluster più grande, B , aggiorniamo il riferimento *cluster* presente in quella posizione in modo che punti a B . Quindi, l'operazione `union(p, q)` viene eseguita in un tempo $O(\min(n_p, n_q))$, dove n_p e n_q sono, rispettivamente, le cardinalità dei cluster contenenti p e q . Questo tempo è chiaramente $O(n)$, se n è il numero di elementi dell'universo partizionato. Vedremo, però, ora un'analisi ammortizzata che mostra come questa implementazione sia ben migliore di quanto appaia dall'analisi del caso peggiore.

Proposizione 14.26: *Usando l'implementazione della partizione basata su sequenze, l'esecuzione di una serie di k operazioni `makeCluster`, `union` e `find` in una partizione inizialmente vuota contenente al massimo n elementi richiede un tempo $O(k + n \log n)$.*

Dimostrazione: Usiamo un'analisi ammortizzata e ipotizziamo che un ciber-dollar sia il costo di un'operazione `find`, di un'operazione `makeCluster` o dello spostamento di un oggetto di tipo “posizione” da una sequenza a un’altra durante l'esecuzione di un'operazione `union`. Nel caso di un'operazione `find` o `makeCluster`, addebitiamo un ciber-dollar per l'operazione stessa. Nel caso di un'operazione `union`, ipotizziamo che un ciber-dollar paghi il lavoro, che richiede un tempo costante, speso per confrontare le dimensioni delle due sequenze, e che si debba pagare un ciber-dollar per ogni posizione che deve essere spostata dal cluster più piccolo al cluster più grande. Chiaramente, il ciber-dollar addebitato per ogni operazione `find` e `makeCluster`, insieme al primo ciber-dollar pagato per ogni operazione `union`, generano una spesa totale di k ciber-dollari per le k operazioni.

Consideriamo, allora, il numero di ciber-dollarai addebitati per modificare le posizioni durante le operazioni `union`. L'osservazione importante è che, ogni volta che spostiamo una posizione da un cluster (più piccolo) a un altro (più grande), la posizione si trova ad appartenere a un cluster (il più grande) la cui dimensione è, alla fine dell'operazione, almeno doppia della dimensione che aveva il cluster (il più piccolo) a cui apparteneva, in partenza, la posizione spostata. Quindi, ogni posizione viene spostata da un cluster a un altro un numero di volte che, al massimo, è pari a $\log n$, per cui dobbiamo pagare al massimo $O(\log n)$ volte per ogni posizione. Avendo ipotizzato che la partizione sia inizialmente vuota, la serie di operazioni può fare riferimento al massimo a un numero $O(n)$ di elementi diversi, per cui il tempo totale necessario per spostare tali elementi durante le operazioni `union` è $O(n \log n)$. ■

Un'implementazione di partizione basata su albero *

Una struttura dati alternativa per rappresentare una partizione usa una collezione di alberi per memorizzare gli n elementi, associando ciascun albero a un diverso cluster. Nello specifico, implementiamo ogni albero con una struttura dati concatenata, i cui nodi sono gli oggetti di tipo "posizione", come si può vedere nella Figura 14.26. Consideriamo che ciascuna posizione p sia un nodo con una variabile di esemplare, `element`, che fa riferimento al proprio elemento, x , e una variabile di esemplare, `parent`, che fa riferimento al nodo genitore. Per convenzione, se p è la *radice* dell'albero a cui appartiene, assegniamo al suo campo `parent` un riferimento a se stessa.

Con questa struttura dati per la partizione, l'operazione `find(p)` viene eseguita risalendo dalla posizione p fino alla radice del suo albero, cosa che richiede un tempo $O(n)$ nel caso peggiore. L'operazione `union(p, q)` può essere implementata facendo in modo che uno dei due alberi diventi un sottoalbero dell'altro: per prima cosa si trovano le due radici, poi in un ulteriore tempo $O(1)$ si fa in modo che il riferimento `parent` di una delle radici punti all'altra radice. La Figura 14.27 mostra un esempio di entrambe le operazioni.

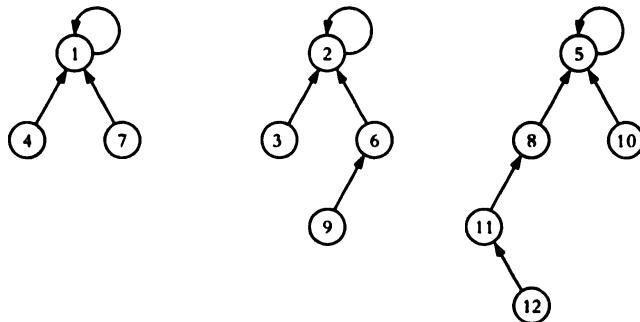


Figura 14.26: Implementazione mediante alberi di una partizione costituita da tre cluster: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$ e $C = \{5, 8, 10, 11, 12\}$.

A un primo esame questa implementazione non sembra in alcun modo migliore della struttura dati basata su sequenze, ma, per renderla più veloce, possiamo aggiungere queste due semplici tecniche euristiche.

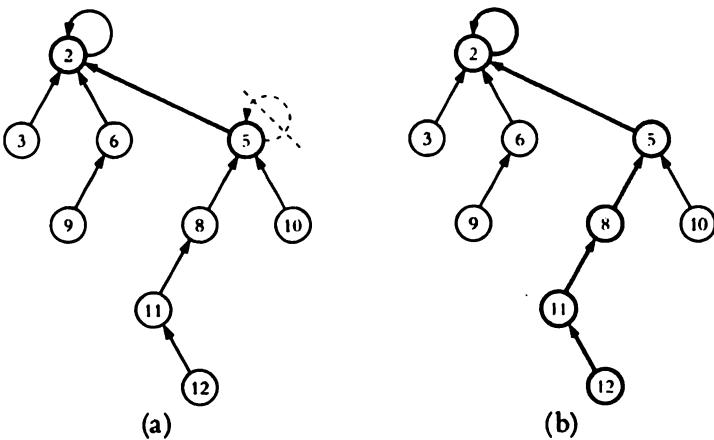


Figura 14.27: Implementazione mediante alberi di una partizione: (a) operazione $\text{union}(p, q)$; (b) operazione $\text{find}(p)$, dove p indica l'oggetto "posizione" corrispondente all'elemento 12.

Euristica “Unione in base alla dimensione”: (*union-by-size*) In ciascuna posizione p memorizziamo anche il numero di elementi presenti nel sottoalbero avente radice in p . Durante un’operazione `union`, facciamo in modo che la radice del cluster più piccolo diventi un figlio dell’altra radice, poi aggiorniamo la dimensione memorizzata in quest’ultima.

Euristica “Compressione del Percorso”: (*path compression*) Durante un’operazione `find`, in ogni posizione q che viene visitata si assegna la radice come genitore (come illustrato nella Figura 14.28).

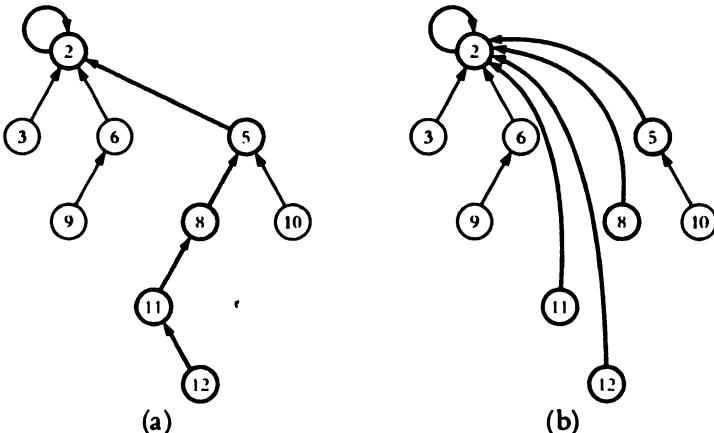


Figura 14.28: Strategia euristica di compressione del percorso: (a) percorso attraversato durante un’operazione `find` che cerca l’elemento 12; (b) albero ristrutturato.

Una proprietà sorprendente di questa struttura dati, quando vengono implementate le due strategie euristiche citate (*union-by-size* e *path compression*), è che l’esecuzione di una serie di k operazioni che coinvolgano n elementi richiede un tempo $O(k \log^* n)$, dove \log^* è

è la cosiddetta “funzione *log-asterisco*” (*log-star*), che è l’ inversa della funzione *torre-di-due* (*tower-of-twos*). Dal punto di vista intuitivo, $\log^* n$ è il numero di applicazioni consecutive al numero n della funzione “logaritmo in base 2” che sono consentite prima che il numero risultante diventi minore di 2. La Tabella 14.4 mostra alcuni valori, come esempio.

Tabella 14.4: Alcuni valori di $\log^* n$ e i valori critici per la sua funzione inversa.

n minimo	2	$2^2 = 4$	$2^{2^2} = 16$	$2^{2^{2^2}} = 65536$	$2^{2^{2^{2^2}}} = 2^{65536}$
$\log^* n$	1	2	3	4	5

Proposizione 14.27: Usando l’implementazione della partizione basata su alberi, con le due strategie euristiche (*union-by-size* e *path compression*), l’ esecuzione di una serie di k operazioni *make-Cluster*, *union* e *find* in una partizione inizialmente vuota contenente al massimo n elementi richiede un tempo $O(k \log^* n)$.

Anche se l’ analisi di questa struttura è piuttosto complessa, la sua implementazione è relativamente semplice, per cui concludiamo il capitolo con la sua implementazione in Java, nel Codice 14.18.

Codice 14.18: Implementazione in Java di una classe *Partition* che usa le strategie euristiche *union-by-size* e *path compression*. Abbiamo omesso soltanto il metodo *validate*, per motivi di spazio.

```

1  /** Una struttura Union-Find per gestire una partizione. */
2  public class Partition<E> {
3      //----- classe Locator annidata -----
4      private class Locator<E> implements Position<E> {
5          public E element;
6          public int size;
7          public Locator<E> parent;
8          public Locator(E elem) {
9              element = elem;
10             size = 1;
11             parent = this; // convenzione per il leader del cluster
12         }
13         public E getElement() { return element; }
14     } //---- fine della classe Locator annidata -----
15     /** Crea un nuovo cluster con l’elemento e e restituisce la sua posizione. */
16     public Position<E> makeCluster(E e) {
17         return new Locator<E>(e);
18     }
19     /**
20      * Cerca il cluster contenente l’elemento identificato dalla Position p
21      * e restituisce la Position del leader del cluster trovato.
22      */
23     public Position<E> find(Position<E> p) {
24         Locator<E> loc = validate(p);
25         if (loc.parent != loc)
26             loc.parent = (Locator<E>) find(loc.parent); // cambia parent dopo ricorsione
27         return loc.parent;
28     }
29     /** Fonde i cluster che hanno gli elementi delle posizioni p e q (se diversi). */
30     public void union(Position<E> p, Position<E> q) {
31         Locator<E> a = (Locator<E>) find(p);
32         Locator<E> b = (Locator<E>) find(q);

```

```

33     if (a != b)
34         if (a.size > b.size) {
35             b.parent = a;
36             a.size += b.size;
37         } else {
38             a.parent = b;
39             b.size += a.size;
40         }
41     }
42 }
```

14.8 Esercizi

Riepilogo e approfondimento

- R-14.1 Disegnare un grafo G semplice e non orientato che abbia 12 vertici, 18 lati e 3 componenti connessi.
- R-14.2 Se G è un grafo semplice e non orientato con 12 vertici e 3 componenti connessi, qual è il massimo numero di lati che può avere?
- R-14.3 Disegnare una rappresentazione mediante matrice di adiacenze del grafo non orientato della Figura 14.1.
- R-14.4 Disegnare una rappresentazione mediante lista di adiacenze del grafo non orientato della Figura 14.1.
- R-14.5 Disegnare un grafo orientato, semplice e连通的, che abbia 8 vertici e 16 lati, tale che il grado entrante e il grado uscente di ciascun vertice sia 2. Dimostrare che esiste un unico ciclo (non semplice) che contiene tutti i lati del grafo: in pratica, questo significa che è possibile percorrere tutti i lati seguendo la loro direzione senza mai staccare la penna dal foglio nel quale è disegnato il grafo. Un tale ciclo viene chiamato *percorso di Euler* (*Euler tour*).
- R-14.6 Supponiamo di rappresentare con una struttura a lista di lati un grafo G avente n vertici e m lati. Perché, in tal caso, il metodo `insertVertex` viene eseguito in un tempo $O(1)$, mentre il metodo `removeVertex` richiede un tempo $O(m)$?
- R-14.7 Descrivere, mediante pseudocodice, come si possa eseguire l'operazione `insertEdge(u , v , x)` in un tempo $O(1)$ usando la rappresentazione mediante matrice di adiacenze.
- R-14.8 Ripetere l'Esercizio R-14.7 per la rappresentazione a lista di adiacenze, così come è stata descritta in questo capitolo.
- R-14.9 Si può evitare la presenza della lista di lati E nella rappresentazione mediante matrice di adiacenze, pur ottenendo le prestazioni temporali descritte nella Tabella 14.1? Perché, oppure perché no?
- R-14.10 Si può evitare la presenza della lista di lati E nella rappresentazione mediante lista di adiacenze, pur ottenendo le prestazioni temporali descritte nella Tabella 14.3? Perché, oppure perché no?
- R-14.11 In quali dei seguenti casi usereste una struttura a matrice di adiacenze e in quali una struttura a lista di adiacenze? Perché?
- Il grafo ha 10000 vertici e 20000 lati, ed è importante usare la minor quantità di spazio possibile.

- b. Il grafo ha 10000 vertici e 20000000 di lati, ed è importante usare la minor quantità di spazio possibile.
- c. La struttura deve eseguire il metodo `getEdge(u, v)` nel più breve tempo possibile, indipendentemente dallo spazio utilizzato.

R-14.12 Per verificare che tutti i suoi lati che non appartengono all'albero DFS siano lati di tipo *back*, ridisegnare il grafo della Figura 14.8b in modo che i lati che fanno parte dell'albero DFS vengano rappresentati con linee a tratto continuo e orientate verso il basso, come nella raffigurazione standard di un albero, mentre tutti gli altri lati siano disegnati con linee tratteggiate.

R-14.13 Spiegare perché, in un grafo semplice avente n vertici che sia rappresentato con una struttura a matrice di adiacenze, l'attraversamento DFS viene eseguito in un tempo $O(n^2)$.

R-14.14 Un grafo semplice e non orientato si dice *completo* se contiene un lato tra ciascuna coppia di vertici distinti. Che aspetto ha l'albero DFS di un grafo completo?

R-14.15 Sulla base della definizione di grafo completo data nell'Esercizio R-14.14, che aspetto ha l'albero BFS di un grafo completo?

R-14.16 Sia G un grafo non orientato i cui vertici sono identificati dai numeri interi che vanno da 1 a 8, caratterizzati dalle adiacenze date dalla tabella seguente:

Vertici	Vertici adiacenti
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Ipotizzare che, durante un attraversamento di G , i vertici adiacenti di un determinato vertice vengano esaminati nello stesso ordine in cui sono stati elencati nella tabella.

- a. Disegnare G .
- b. Scrivere i vertici di G nell'ordine in cui vengono visitati durante un attraversamento DFS che parta dal vertice 1.
- c. Scrivere i vertici di G nell'ordine in cui vengono visitati durante un attraversamento BFS che parta dal vertice 1.

R-14.17 Bob ama le lingue straniere e vuole progettare il suo piano di studi per i prossimi anni. È interessato a seguire nove insegnamenti di lingue: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141 e LA169. Dati questi prerequisiti per ciascun insegnamento:

- LA15: (nessuno)
- LA16: LA15
- LA22: (nessuno)

- LA31: LA15
 - LA32: LA16, LA31
 - LA126: LA22, LA32
 - LA127: LA16
 - LA141: LA22, LA16
 - LA169: LA32

in che ordine Bob può seguire questi insegnamenti, rispettandone i prerequisiti?

R-14.18 Individuare un ordinamento topologico per il grafo orientato che, nella Figura 14.3d, è stato disegnato a tratto continuo.

R-14.19 Disegnare la chiusura transitiva del grafo orientato della Figura 14.2.

R-14.20 Se i vertici del grafo della Figura 14.11 fossero ordinati come (JFK, LAX, MIA, BOS, ORD, SFO, DFW), in quale ordine verrebbero aggiunti i lati alla chiusura transitiva durante l'esecuzione dell'algoritmo di Floyd-Warshall?

R-14.21 Quanti lati ci sono nella chiusura transitiva di un grafo costituito da un semplice percorso orientato con n vertici?

R-14.22 Dato un albero binario completo T con n nodi, avente come radice un nodo qualsiasi, considerare un grafo orientato \tilde{G} avente come vertici i nodi di T e un lato per ogni coppia genitore-figlio presente in T , orientato in \tilde{G} dal genitore al figlio. Dimostrare che la chiusura transitiva di \tilde{G} ha $O(n \log n)$ lati.

R-14.23 Disegnare un grafo semplice, connesso e pesato, con 8 vertici e 16 lati, con pesi tutti diversi. Scegliere un vertice qualsiasi come "partenza" e illustrare l'esecuzione dell'algoritmo di Dijkstra su tale grafo.

R-14.24 Spiegare come si possa modificare lo pseudocodice che descrive l'algoritmo di Dijkstra per gestire il caso in cui il grafo sia orientato e si vogliano calcolare le lunghezze minime dei percorsi orientati che vanno dal vertice origine a qualsiasi altro vertice.

R-14.25 Disegnare un grafo semplice,连通的, pesato e non orientato, con 8 vertici e 16 lati, con pesi tutti diversi. Illustrare l'esecuzione su tale grafo dell'algoritmo di Prim-Jarník per il calcolo di un albero ricoprente minimo.

R-14.26 Ripetere l'esercizio precedente per l'algoritmo di Kruskal.

R-14.27 In un lago sono presenti otto isolette e lo stato vuole costruire sette ponti per collegarle, in modo che ogni isola possa essere raggiunta da qualsiasi altra percorrendo uno o più ponti. Il costo della costruzione di un ponte è proporzionale alla sua lunghezza e le distanze tra ciascuna coppia di isole è riportata nella tabella seguente.

Trovare quali sono i ponti da costruire per rendere minimo il costo di costruzione complessivo.

- R-14.28 Descrivere il significato delle convenzioni grafiche usate nella Figura 14.9 che illustra un attraversamento DFS. Che significato ha lo spessore delle linee? Cosa significano le frecce? E i tratteggi?
- R-14.29 Ripetere l'Esercizio R-14.28 per la Figura 14.8 che illustra un attraversamento DFS in un grafo orientato.
- R-14.30 Ripetere l'Esercizio R-14.28 per la Figura 14.10 che illustra un attraversamento BFS.
- R-14.31 Ripetere l'Esercizio R-14.28 per la Figura 14.11 che illustra un'esecuzione dell'algoritmo di Floyd-Warshall.
- R-14.32 Ripetere l'Esercizio R-14.28 per la Figura 14.13 che illustra un'esecuzione dell'algoritmo di ordinamento topologico.
- R-14.33 Ripetere l'Esercizio R-14.28 per le Figure 14.15 e 14.16 che illustrano un'esecuzione dell'algoritmo di Dijkstra.
- R-14.34 Ripetere l'Esercizio R-14.28 per le Figure 14.20 e 14.21 che illustrano un'esecuzione dell'algoritmo di Prim-Jarník.
- R-14.35 Ripetere l'Esercizio R-14.28 per le Figure 14.22, 14.23 e 14.24 che illustrano un'esecuzione dell'algoritmo di Kruskal.
- R-14.36 George afferma di aver individuato un modo veloce per effettuare la compressione dei percorsi, a partire dalla posizione p , in una struttura che rappresenta una partizione. Si inserisce p in una lista L e si seguono i riferimenti verso i genitori. Ogni volta che si incontra una nuova posizione, q , si aggiunge q a L e si aggiorna il riferimento al genitore in tutti i nodi di L in modo che puntino al genitore di q . Dimostrare che l'algoritmo di George richiede un tempo d'esecuzione $\Omega(h^2)$ per un percorso di lunghezza h .

Creatività

- C-14.37 Implementare in Java il metodo `removeEdge(e)` per la realizzazione di grafo mediante mappa di adiacenze presentata nel Paragrafo 14.2.5, facendo in modo che funzioni tanto per grafi orientati quanto per grafi non orientati. Il metodo deve essere eseguito in un tempo $O(1)$.
- C-14.38 Supponiamo di voler rappresentare un grafo G avente n vertici usando una struttura a lista di lati, nell'ipotesi di identificare i vertici con i numeri interi appartenenti all'insieme $\{0, 1, \dots, n - 1\}$. Descrivere come si possa implementare la collezione E in modo che consenta l'esecuzione del metodo `getEdge(u, v)` in un tempo $O(\log n)$. Come si implementerebbe il metodo in questo caso?
- C-14.39 Sia T l'albero ricoprente, avente come radice il vertice di partenza, che viene generato dall'attraversamento in profondità di un grafo G connesso e non orientato. Spiegare perché ogni lato di G che non appartiene a T va da un vertice di T a uno dei suoi antenati in T , cioè è un *lato back*.
- C-14.40 La soluzione riportata nel Codice 14.6 per risolvere il problema della ricostruzione di un percorso da u a v potrebbe essere resa più efficiente, in pratica, se la procedura DFS terminasse non appena raggiunto v per la prima volta. Descrivere come vada modificato il codice per implementare questa ottimizzazione.

- C-14.41 Sia G un grafo non orientato avente n vertici e m lati. Descrivere un algoritmo che in un tempo $O(n + m)$ attraversi ogni lato di G esattamente una volta in ciascuna delle due possibili direzioni.
- C-14.42 Implementare un algoritmo che restituisce un ciclo presente in un grafo orientato \tilde{G} , se ne esiste almeno uno.
- C-14.43 Scrivere un metodo, `components(G)`, che, dato un grafo non orientato G , restituisca una mappa che associa ciascun vertice a un numero intero che serve a identificare il componente连通 a cui appartiene. Detto in altri termini, due vertici devono essere associati allo stesso numero se e solo se appartengono allo stesso componente连通.
- C-14.44 Dato un labirinto, diciamo che è *costruito correttamente* se: esiste un percorso che va dal punto di partenza al punto di arrivo; l'intero labirinto è raggiungibile dal punto di partenza; non sono presenti cicli nel labirinto. Dato un labirinto, disegnato in una griglia $n \times n$, come si può determinare se è stato costruito correttamente? Qual è il tempo d'esecuzione dell'algoritmo progettato?
- C-14.45 Le reti di calcolatori non dovrebbero avere singoli punti di rottura, cioè vertici della rete che la possano rendere non più connessa in caso di guasto di un unico vertice. Diciamo che un grafo G non orientato e connesso è *biconnesso* se non contiene alcun vertice la cui rimozione da G dividerebbe G in due o più componenti connessi. Progettare un algoritmo che, aggiungendo al massimo n lati a un grafo G connesso, avente $n \geq 3$ vertici e $m \geq n - 1$ lati, garantisca che G sia biconnesso. L'algoritmo deve poter essere eseguito in un tempo $O(n + m)$.
- C-14.46 Spiegare perché, in relazione all'albero BFS costruito per un grafo non orientato, tutti i lati del grafo che non appartengono all'albero sono lati *cross*.
- C-14.47 Spiegare perché, in relazione all'albero BFS costruito per un grafo orientato, tutti i lati del grafo che non appartengono all'albero non sono lati *forward*.
- C-14.48 Dimostrare che, se T è l'albero BFS di un grafo connesso G , allora, per ogni vertice v di livello i , il percorso che va, in T , da s a v ha i lati, mentre ogni altro percorso, in G , da s a v ha almeno i lati.
- C-14.49 Dimostrare la Proposizione 14.16.
- C-14.50 Implementare l'algoritmo BFS in modo che, per memorizzare i vertici che sono stati scoperti e conservarli fino al momento in cui si debbano prendere in esame i vertici a loro adiacenti, utilizzi una coda FIFO, invece di procedere livello per livello.
- C-14.51 Un grafo G è *bipartito* se i suoi vertici possono essere partizionati in due insiemi X e Y tali che ogni lato di G abbia un vertice terminale in X e l'altro in Y . Progettare e analizzare un algoritmo efficiente che determini se un grafo G non orientato sia bipartito (senza conoscere a priori gli insiemi X e Y).
- C-14.52 Un *percorso di Eulero* in un grafo orientato \tilde{G} avente n vertici e m lati è un ciclo che percorre ogni lato di \tilde{G} esattamente una volta secondo la direzione del suo orientamento. Tale percorso esiste sempre se \tilde{G} è connesso e, in ogni vertice di \tilde{G} , il grado entrante è uguale al grado uscente. Descrivere un algoritmo che, in un tempo $O(n + m)$, trovi un percorso di Eulero per un grafo orientato \tilde{G} avente tali caratteristiche.
- C-14.53 La compagnia telefonica RT&T ha una rete di n stazioni di commutazione connesse mediante m linee di trasmissione ad alta velocità. Il telefono di ciascun cliente è connesso direttamente a una stazione di commutazione. Gli ingegneri

di RT&T hanno sviluppato un prototipo di video-telefono che consente a due clienti di vedersi durante una telefonata. Perché la qualità delle immagini sia accettabile, però, il numero di linee utilizzate per trasmettere i segnali video tra i due partecipanti alla comunicazione non può essere superiore a 4. Supponendo che la rete RT&T sia rappresentata da un grafo, progettare un algoritmo efficiente che calcoli, per ciascuna stazione, l'insieme di stazioni della rete che possono essere raggiunte usando non più di 4 linee di collegamento.

- C-14.54 Il ritardo presente sul segnale audio durante una telefonata a lunga distanza si può determinare moltiplicando un piccolo valore costante per il numero di linee di comunicazione che si trovano, nella rete telefonica, tra il chiamante e il chiamato. Supponiamo che la rete della compagnia telefonica RT&T sia un albero. Gli ingegneri di RT&T vogliono calcolare il massimo ritardo possibile che può caratterizzare una telefonata a lunga distanza. Dato un albero T , il *diametro* di T è la lunghezza del più lungo percorso presente in T tra due qualsiasi nodi di T . Descrivere un algoritmo efficiente per calcolare il diametro di T .
- C-14.55 La Tamarindo University e molte altre scuole nel mondo sono coinvolte in un progetto multimediale congiunto. È stata costruita una rete di calcolatori per connettere tali scuole usando linee di comunicazione che costituiscono un albero. Si è, poi, deciso di installare un *file server* in una delle scuole, in modo da poter condividere dati tra tutte le scuole partecipanti al progetto. Dato che il tempo di trasmissione lungo una linea è dominato dal tempo di attivazione (*setup*) e sincronizzazione della linea, il costo di un trasferimento di dati è proporzionale al numero di linee utilizzate. Quindi, è preferibile scegliere, per il *file server*, una collocazione "centrale" nella rete. Dato un albero T e un nodo v di T , l'*eccentricità* di v è la lunghezza del più lungo percorso che va da v a un altro nodo di T . Un nodo di T che abbia eccentricità minima è chiamato *centro* di T .
- Progettare un algoritmo efficiente che, dato un albero T con n nodi, ne individui un centro.
 - Il centro è sempre unico? Se non lo è, quanti diversi centri possono esistere in un albero?
- C-14.56 Diciamo che un grafo orientato aciclico \tilde{G} avente n vertici è *compatto* se esiste un modo per etichettare i suoi vertici con i numeri interi che vanno da 0 a $n - 1$ tale che \tilde{G} contenga il lato (i, j) se e solo se $i < j$, per ogni valore di i e j nell'intervallo $[0, n - 1]$. Progettare un algoritmo che, in un tempo $O(n^2)$, determini se \tilde{G} è compatto.
- C-14.57 Sia \tilde{G} un grafo orientato e pesato, con n vertici. Progettare una variante dell'algoritmo di Floyd-Warshall che, in un tempo $O(n^3)$, calcoli le lunghezze dei percorsi più brevi che vanno da ciascun vertice a qualsiasi altro vertice.
- C-14.58 Progettare un algoritmo efficiente per trovare, in un grafo orientato aciclico e pesato \tilde{G} , un percorso orientato di *lunghezza massima* che vada da un vertice s a un vertice t . Specificare la rappresentazione del grafo che si intende utilizzare e qualsiasi struttura dati ausiliaria utilizzata dall'algoritmo, analizzandone le prestazioni temporali asintotiche.
- C-14.59 Un insieme indipendente (*independent set*) di un grafo non orientato $G = (V, E)$ è un sottoinsieme I di V tale che in I non esistano due vertici adiacenti: detto in

altri termini, se u e v appartengono a I , allora (u, v) non appartiene a E . Un *insieme indipendente massimale* (*maximal independent set*) M è un insieme indipendente tale che, se si aggiungesse a M un ulteriore vertice, M non sarebbe più un insieme indipendente. Ogni grafo ha almeno un insieme indipendente massimale (siete in grado di spiegare perché? questa domanda non fa parte dell'esercizio, ma vale la pena rifletterci). Progettare un algoritmo efficiente che individui un insieme indipendente massimale per un grafo G . Quali sono le sue prestazioni temporali?

- C-14.60 Fornire un esempio di grafo semplice con n vertici che richieda un tempo $\Omega(n^2 \log n)$ per l'esecuzione dell'algoritmo di Dijkstra implementato in modo che usi uno heap.
- C-14.61 Fornire un esempio di grafo orientato e pesato con lati il cui peso può essere anche negativo, ma privo di cicli di peso negativo, tale che l'esecuzione dell'algoritmo di Dijkstra calcoli in modo errato la distanza di qualche vertice dal vertice di partenza s .
- C-14.62 La nostra implementazione del metodo `shortestPathLengths`, presentata nel Codice 14.13, sfrutta un valore numerico speciale come "infinito" per rappresentare la distanza massima dall'origine di quei vertici per i quali non è ancora nota la raggiungibilità dall'origine. Implementare in modo diverso quel metodo in modo che non usi quella sentinella: evitare di aggiungere alla coda prioritaria i vertici, diversi dall'origine, finché non sia evidente che sono raggiungibili dall'origine.
- C-14.63 Considerare la seguente strategia greedy per trovare un percorso di peso minimo dal vertice $start$ al vertice $goal$ in un grafo connesso.

1. Inizializza $path$ a $start$.
2. Inizializza l'insieme $visited$, contenente i vertici visitati, a $\{start\}$.
3. Se $start = goal$, restituisci $path$ e termina, altrimenti procedi.
4. Trova il lato $(start, v)$ di peso minimo tale che v sia adiacente a $start$ e v non sia stato visitato.
5. Aggiungi v a $path$.
6. Aggiungi v a $visited$.
7. Assegna v a $start$ e torna al punto 3.

Questa strategia greedy trova sempre un percorso di peso minimo tra $start$ e $goal$? Spiegare intuitivamente perché funziona, oppure trovare un controesempio che dimostri che non funziona.

- C-14.64 Dimostrare che, se in un grafo connesso e pesato G tutti i pesi sono diversi, allora esiste uno e un solo albero ricoprente minimo per G .
- C-14.65 Un vecchio metodo per individuare un MST, chiamato *algoritmo di Baruvka*, funziona nel modo seguente, dato un grafo G avente n vertici e m lati con pesi tutti diversi:

```

sia  $T$  un sotto-grafo di  $G$  che contiene inizialmente soltanto i vertici di  $V$ 
while  $T$  ha un numero di lati inferiore a  $n - 1$  do
    for ogni componente连通的  $C_i$  di  $T$  do
        trova il lato  $(u, v)$  di peso minimo in  $E$  con  $u$  in  $C_i$  e  $v$  non in  $C_i$ 
        aggiungi  $(u, v)$  a  $T$  (a meno che non sia già in  $T$ )
    return  $T$ 
```

Dimostrare che questo algoritmo è corretto e che viene eseguito in un tempo $O(m \log n)$.

C-14.66 Sia G un grafo con n vertici e m lati tale che tutti i pesi dei lati siano numeri interi appartenenti all'intervallo $[1, n]$. Progettare un algoritmo che trovi un albero ricoprente minimo per G in un tempo $O(m \log^* n)$.

C-14.67 Considerare lo schema di una rete telefonica, che è un grafo G i cui vertici rappresentano le centrali di commutazione e i cui lati rappresentano le linee di comunicazione che collegano coppie di centrali. I lati sono pesati con i valori della loro ampiezza di banda (*bandwidth*) e l'ampiezza di banda di un percorso è uguale alla minima ampiezza di banda dei lati che lo costituiscono. Progettare un algoritmo che, data una rete e due centrali di commutazione a e b , trovi la massima ampiezza di banda di un percorso tra a e b .

C-14.68 La NASA (National Aeronautics and Space Administration) vuole collegare n stazioni distribuite sul territorio nazionale usando opportuni canali di comunicazione. Ogni coppia di stazioni ha una diversa ampiezza di banda disponibile, nota a priori. La NASA vuole individuare $n - 1$ canali (cioè il minimo numero possibile) in modo tale che tutte le stazioni siano collegate e che l'ampiezza di banda totale (definita come la somma delle singole ampiezze di banda dei canali) sia massima. Progettare un algoritmo efficiente per risolvere questo problema e determinarne la complessità temporale nel caso peggiore. Considerare il grafo pesato $G = (V, E)$, dove V è l'insieme delle stazioni e E è l'insieme dei canali tra le stazioni. Definire il peso $w(e)$ del lato e in E come l'ampiezza di banda del canale corrispondente.

C-14.69 Nel Castello di Asymptopia c'è un labirinto e in ogni corridoio del labirinto c'è un sacco di monete d'oro, con quantità diverse. A un nobile cavaliere, Sir Paul, verrà data l'opportunità di vagare nel labirinto, raccogliendo sacchi d'oro. Potrà entrare nel labirinto soltanto passando dalla porta "ENTER" e uscire da un'altra porta, "EXIT". Mentre si troverà nel labirinto non potrà tornare sui propri passi: ogni corridoio del labirinto ha una freccia disegnata sul pavimento, che indica la direzione in cui lo si può percorrere e che dovrà essere seguita da Sir Paul. Il labirinto non presenta "cicli". Progettare un algoritmo che, data una mappa del labirinto che riporta la quantità di monete d'oro presenti in ciascun sacco, aiuti Sir Paul a raccogliere la massima quantità di monete d'oro.

C-14.70 Si dispone di un *orario*, che riporta:

- Un insieme \mathcal{A} di n aeroporti e, per ciascun aeroporto a in \mathcal{A} , il tempo minimo di interconnessione, $c(a)$, necessario per passare da un volo a un altro, essendo in transito.
- Un insieme \mathcal{F} di m voli e, per ciascun volo f in \mathcal{F} , le seguenti informazioni:
 - Aeroporto di partenza, $a_1(f)$ in A
 - Aeroporto di arrivo, $a_2(f)$ in A
 - Orario di partenza, $t_1(f)$
 - Orario di arrivo, $t_2(f)$

Descrivere un algoritmo efficiente per risolvere il problema della pianificazione dei voli (*flight scheduling*). In questo problema, sono dati due aeroporti, a e b , e un istante di tempo t , e si vuole individuare una sequenza di voli che consenta di arrivare il più presto possibile in b partendo da a non prima dell'istante t , rispettando i tempi minimi di interconnessione degli aeroporti intermedi. Qual è il tempo d'esecuzione dell'algoritmo progettato, in funzione di n e di m ?

C-14.71 Sia dato un grafo orientato \tilde{G} con n vertici e sia M la matrice di adiacenze, $n \times n$, di \tilde{G} .

- Definiamo il prodotto di M con se stessa (M^2) in questo modo, con $1 \leq i \leq j \leq n$:

$$M^2(i, j) = M(i, 1) \odot M(1, j) \oplus \cdots \oplus M(i, n) \odot M(n, j),$$

dove il simbolo \oplus è l'operatore booleano or e il simbolo \odot è l'operatore booleano and. Data questa definizione, cosa implica la relazione $M^2(i, j) = 1$ per i vertici i e j ? E $M^2(i, j) = 0$?

- Indichiamo con M^4 il prodotto di M^2 con se stessa. Che significato hanno le singole componenti di M^4 ? E quelle di $M^5 = (M^4)(M)$? In generale, che informazione è contenuta nella matrice M^p ?
- Supponiamo, ora, che il grafo \tilde{G} sia pesato e che siano valide le seguenti ipotesi:
 - per $1 \leq i \leq n$, $M(i, i) = 0$.
 - per $1 \leq i < j \leq n$, $M(i, j) = \text{peso}(i, j)$ se (i, j) appartiene a E .
 - per $1 \leq i < j \leq n$, $M(i, j) = \infty$ se (i, j) non appartiene a E .

Ancora, definiamo M^2 , con $1 \leq i < j \leq n$, in questo modo:

$$M^2(i, j) = \min\{M(i, 1) + M(1, j), \dots, M(i, n) + M(n, j)\}.$$

Se $M^2(i, j) = k$, cosa possiamo concludere in merito alla relazione esistente tra i vertici i e j ?

C-14.72 Karen afferma di aver scoperto un modo nuovo per effettuare la compressione del percorso, a partire da una posizione p , in una struttura dati che implementa una partizione di tipo union/find mediante alberi. Per prima cosa inserisce in un insieme S tutte le posizioni che si trovano lungo il percorso che va da p alla radice. Poi, analizza tutti gli elementi di S e in ciascuna di tali posizioni modifica il valore del riferimento parent in modo che punti al genitore del suo genitore (ricordando che, nella radice, il riferimento parent punta alla radice stessa.). Se questa fase modifica il valore del riferimento parent di una qualsiasi posizione, si ripete l'intera procedura, continuando a ripeterla finché non viene eseguita un'intera scansione di S senza che si modifichi il riferimento parent di nessuna posizione. Dimostrare che l'algoritmo di Karen è corretto e analizzare il suo tempo d'esecuzione per un percorso di lunghezza h .

Progettazione

- P-14.73 Usare una matrice di adiacenze per implementare una classe che fornisca supporto a un tipo di dato astratto semplificato per grafi, che non contenga alcun metodo di modifica del grafo stesso. La classe deve avere un costruttore che genera un grafo corrispondente ai due parametri ricevuti: un contenitore, V , con gli elementi che vanno posti nei vertici e un altro, E , che è un contenitore di coppie di elementi contenuti nei vertici, che siano i vertici terminali di un lato.
- P-14.74 Implementare il tipo di dato astratto "grafo semplificato" descritto nell'Esercizio P-14.73 usando una struttura a lista di lati.
- P-14.75 Implementare il tipo di dato astratto "grafo semplificato" descritto nell'Esercizio P-14.73 usando una struttura a lista di adiacenze.
- P-14.76 Estendere la classe dell'Esercizio P-14.75 in modo che fornisca anche il supporto ai metodi di modifica del tipo di dato astratto "grafo".
- P-14.77 Progettare un esperimento che confronti la generazione della chiusura transitiva di un grafo orientato effettuata mediante ripetuti traversamenti DFS oppure invocando l'algoritmo di Floyd-Warshall.
- P-14.78 Progettare un'implementazione in Java dell'algoritmo di Prim-Jarník per la costruzione di un albero ricoprente minimo in un grafo.
- P-14.79 Eseguire un esperimento che confronti i due algoritmi di generazione di MST presentati in questo capitolo (Kruskal e Prim-Jarník): sviluppare un insieme di esperimenti che verifichi in modo esauriente i tempi d'esecuzione di questi algoritmi, usando grafi generati casualmente.
- P-14.80 Per costruire un *labirinto* si può partire da una griglia $n \times n$ progettata in modo tale che ogni cella della griglia sia delimitata da quattro muri di lunghezza unitaria. Poi, si rimuovono due muri di lunghezza unitaria posti ai bordi della griglia, per rappresentare l'ingresso del labirinto e la sua uscita. A ogni muro unitario rimasto che non si trovi sul bordo, assegniamo un valore casuale e creiamo un grafo, G , chiamato *duale*, tale che G contenga un vertice per ogni cella della griglia e un lato a collegare i vertici di due celle adiacenti se e solo se queste condividono un muro divisorio. Il peso di ciascun lato è il peso del muro corrispondente (il valore generato casualmente in precedenza). Costruiamo il labirinto cercando un albero ricoprente minimo T per G e rimuovendo tutti i muri corrispondenti ai lati di T . Scrivere un programma che usi questo algoritmo per generare labirinti e, poi, li risolva. Come minimo, il programma deve visualizzare il labirinto, anche se è decisamente preferibile che visualizzi anche una sua soluzione, cioè un percorso che vada dall'entrata all'uscita.
- P-14.81 Scrivere un programma che costruisca la tabella di instradamento (*routing table*) per i nodi di una rete di calcolatori, basata sull'algoritmo di instradamento a percorso minimo (*shortest-path routing*), nel quale la lunghezza di un percorso è misurata dal conteggio dei "salti" (*hop*), cioè dal numero di lati del percorso stesso. I dati in ingresso per questo problema sono le informazioni di connettività per tutti i nodi della rete, come in questo esempio (dove i nodi sono identificati mediante il loro indirizzo IP):

241.12.31.14: 241.12.31.15 241.12.31.18 241.12.31.19

che indica che tre nodi della rete sono connessi al nodo 241.12.31.14, cioè tre nodi si trovano a distanza unitaria (un solo *hop*) da quel nodo. La tabella di instradamento per il nodo che si trova all'indirizzo A è un insieme di coppie (B, C), ciascuna delle quali afferma che, per instradare un messaggio da A verso

B, il nodo successivo a cui inviare il messaggio (lungo il percorso più breve che va da *A* a *B*) è *C*. Il programma deve visualizzare la tabella di instradamento per ciascun nodo della rete, dato in ingresso un elenco delle connettività dei singoli nodi, cioè un insieme di righe come quella riportata qui.

Note

Il metodo di attraversamento in profondità fa parte della tradizione dell'informatica, ma Hopcroft e Tarjan [46, 87] sono quelli che hanno evidenziato quanto possa essere utile tale algoritmo per risolvere molti problemi diversi relativi ai grafi. Knuth [60] ha discusso il problema dell'ordinamento topologico. Il semplice algoritmo che abbiamo descritto per determinare in un tempo lineare se un grafo orientato sia fortemente connesso è dovuto a Kosaraju. L'algoritmo di Floyd-Warshall è apparso in un lavoro di Floyd [34], basato su un teorema di Warshall [94].

Il primo algoritmo noto per l'individuazione di un albero ricoprente minimo è dovuto a Baruvka [9] e fu pubblicato nel 1926. L'algoritmo di Prim-Jarník è stato pubblicato per la prima volta in lingua ceca da Jarník [51] nel 1920 e in inglese da Prim [79] nel 1957. Kruskal ha pubblicato il suo algoritmo per trovare un MST nel 1956 [63]. Il lettore interessato a studiare più a fondo la storia del problema MST è rimandato al lavoro di Graham e Hell [41]. Attualmente, l'algoritmo asintoticamente più veloce per trovare un MST è un metodo probabilistico dovuto a Karger, Klein e Tarjan [53] che viene eseguito in un tempo atteso $O(m)$. Dijkstra ha pubblicato il suo algoritmo per i percorsi minimi da sorgente singola nel 1959 [30]. Il tempo d'esecuzione dell'algoritmo di Prim-Jarník, così come quello dell'algoritmo di Dijkstra, può effettivamente essere migliorato, portandolo a $O(m + n \log n)$, implementando la coda prioritaria *Q* con una struttura dati più sofisticata come lo "Heap di Fibonacci" [36] o lo "Heap rilassato" [32].

Per apprendere algoritmi utili per disegnare grafi, si invita alla lettura del capitolo del libro di Tamassia e Liotta [85] dedicato a questo argomento, così come del libro di Di Battista, Eades, Tamassia e Tollis [29]. Il lettore interessato ad approfondire lo studio di algoritmi per grafi può consultare i libri di Ahuja, Magnanti e Orlin [7], di Cormen, Leiserson, Rivest e Stein [25], di Mehlhorn [72], di Tarjan [88] e di van Leeuwen [90].

15

Gestione della memoria e B-alberi

15.1 Gestione della memoria

La memoria dei calcolatori è organizzata come una sequenza di *parole* (*word*), ciascuna delle quali è tipicamente costituita da 4, 8 o 16 byte (dipende dal calcolatore). Queste parole di memoria sono numerate da 0 a $N - 1$, dove N è il numero di parole di memoria disponibili nel calcolatore. Il numero associato a ciascuna parola di memoria viene chiamato *indirizzo* in memoria della parola stessa. Quindi, in pratica, la memoria di un computer può essere vista come un gigantesco array di parole di memoria, illustrato nella Figura 15.1.



Figura 15.1: Indirizzi di memoria.

Per poter eseguire programmi e memorizzare informazioni, è necessario che la memoria del calcolatore sia *gestita*, in modo da determinare quali dati debbano essere memorizzati in quali celle della memoria. In questo paragrafo discuteremo delle nozioni fondamentali relative alla gestione della memoria, descrivendo, in particolare, come vengano gestite le varie zone della memoria che viene assegnata a un programma Java quando è messo in esecuzione dal sistema operativo, oltre a ciò che accade quando alcune porzioni della memoria vengono rese libere dal programma, perché non più necessarie, e potenzialmente riutilizzate.

15.1.1 Strutture di tipo stack nella Java Virtual Machine

Un programma Java, dopo essere stato compilato, è costituito da una sequenza di “codici a byte” (detti *byte code*) che svolgono il ruolo di istruzioni “macchina” per un modello di macchina ben definito: la *macchina virtuale di Java* (*Java Virtual Machine*, JVM). La definizione della JVM è il nucleo della definizione dello stesso linguaggio Java: compilando il codice sorgente Java per generare codice a byte per la JVM, invece di generare linguaggio macchina per una specifica CPU, si può rendere il programma eseguibile in qualsiasi calcolatore che disponga di un programma in grado di emulare la JVM.

Le pile (o, come sappiamo dal Capitolo 6, *stack*) hanno importanti applicazioni nell’ambiente di esecuzione (*runtime environment*) dei programmi Java. Un programma Java in esecuzione (o, più precisamente, un *thread* Java in esecuzione, che può essere una parte di un programma *multi-threaded*, con più parti cooperanti gestite dal sistema operativo) ha una propria pila, detta *pila dei metodi di Java* o, più brevemente, *pila di Java* o *pila delle invocazioni* (*Java stack* o *call stack*), che viene usata per tenere traccia delle variabili locali e di altre informazioni importanti relative ai metodi così come vengono invocati durante l’esecuzione del programma (si veda la Figura 15.2).

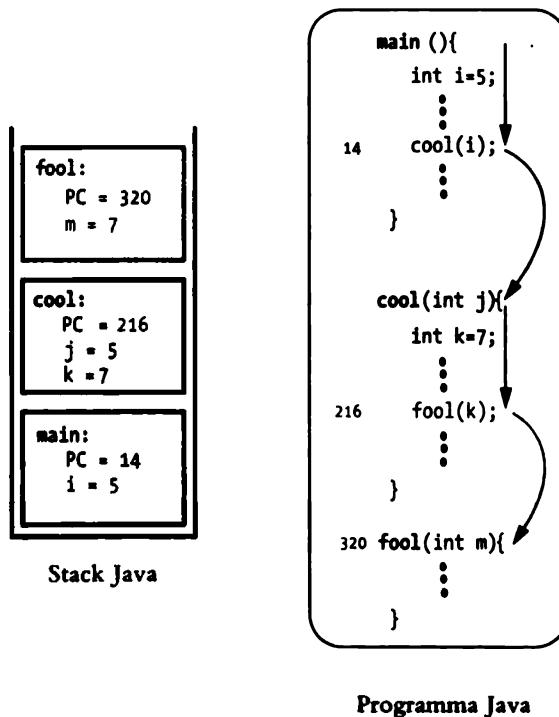


Figura 15.2: Un esempio della *pila dei metodi di Java*: il metodo `fool` è stato invocato dal metodo `cool`, che, a sua volta, era stato invocato dal metodo `main`. Osserviamo i valori del *program counter* (PC, “contatore di programma”) della CPU, dei parametri e delle variabili locali che sono stati memorizzati nei *frame* di invocazione. Quando termina l’esecuzione del metodo `fool`, viene ripresa l’esecuzione del metodo `cool` a partire dall’istruzione 217, valore che si ottiene incrementando di un’unità il valore del *program counter* memorizzato nel suo *frame* presente sulla *pila di Java*.

Entrando più in dettaglio, durante l'esecuzione di un programma Java, la Java Virtual Machine (JVM) gestisce una pila i cui elementi sono i descrittori delle invocazioni di metodi *attivi* in quel momento, cioè non ancora terminati. Questi descrittori vengono solitamente chiamati *frame* (o "dati di attivazione"): un frame relativo a un'invocazione del metodo "fool" contiene i valori delle variabili locali e dei parametri del metodo fool, oltre a quelle informazioni relative al metodo "cool", che ha invocato fool, che sono necessarie perché il flusso dell'esecuzione torni al metodo cool quando l'esecuzione di fool sarà terminata.

Tenere traccia del *program counter*

La JVM usa una variabile speciale, chiamata *program counter* ("contatore di programma"), per tenere traccia dell'indirizzo all'interno del programma dell'enunciato che la JVM sta eseguendo. Quando un metodo "cool" invoca un altro metodo "fool", il valore del program counter viene registrato nel frame dell'invocazione di cool, così la JVM saprà dove riprendere l'esecuzione quando il metodo fool sarà terminato. In cima alla pila di Java c'è sempre il frame relativo al *metodo in esecuzione in quel momento*, che è il metodo che ha il controllo del flusso d'esecuzione. Gli altri elementi presenti nella pila sono frame dei *metodi sospesi*, quei metodi che hanno invocato un altro metodo e, quindi, stanno attendendo che il metodo invocato restituisca loro il controllo, dopo aver terminato la propria esecuzione. L'ordine in cui gli elementi sono disposti nella pila di Java riflette la catena delle invocazioni che lega tra loro i metodi attivi. Quando viene invocato un nuovo metodo, il relativo frame viene inserito in cima alla pila; quando, poi, il metodo termina, il suo frame viene estratto dalla cima della pila e la JVM riprende l'elaborazione prevista dal metodo precedentemente sospeso.

Implementare la ricorsione

Uno dei vantaggi derivanti dall'utilizzo di una pila per implementare l'invocazione dei metodi è la possibilità, che ne deriva, che i programmi usino la *ricorsione*, cioè che un metodo possa invocare se stesso, come visto nel Capitolo 5. In quel capitolo avevamo descritto implicitamente il concetto di pila delle invocazioni (*call stack*) e l'uso dei frame all'interno delle nostre raffigurazioni dei *diagrammi di ricorsione*. È interessante osservare che i primi linguaggi di programmazione, come Cobol e Fortran, non usavano la pila delle invocazioni per implementare le invocazioni di funzioni e procedure, ma oggi tutti i linguaggi di programmazione (comprese le versioni moderne dei linguaggi classici, come Cobol e Fortran) usano una pila delle invocazioni per metodi e procedure, per via dell'eleganza e dell'efficienza consentita dalla ricorsione.

Ogni riquadro di un diagramma di ricorsione corrisponde a un frame nella pila di Java e, in qualsiasi momento, il contenuto della pila di Java corrisponde alla catena di riquadri, a partire dall'invocazione iniziale del metodo fino a quella attuale.

Per illustrare meglio come la pila di Java consenta l'esecuzione di metodi ricorsivi, torniamo all'implementazione, in Java, della classica definizione ricorsiva della funzione fattoriale:

$$n! = n(n - 1)(n - 2) \cdots 1,$$

con il codice originariamente presentato nel Codice 5.1 e il diagramma di ricorsione illustrato nella Figura 5.1. La prima volta che invochiamo il metodo `factorial(n)`, il suo frame sulla pila contiene una variabile locale che memorizza il valore di `n`. Il metodo in-

voca ricorsivamente se stesso per calcolare $(n - 1)!$, cosa che inserisce un nuovo frame sulla pila di Java. A sua volta, questa invocazione ricorsiva invoca se stessa per calcolare $(n - 2)!$, e così via. La catena di invocazioni ricorsive e, quindi, la pila di esecuzione di Java, cresce fino alla dimensione $n + 1$ e, in quel momento, l'invocazione attiva sarà `factorial(0)`, che restituisce 1 senza alcuna ulteriore ricorsione. La pila di esecuzione di Java, quindi, consente l'esistenza simultanea di diverse invocazioni del metodo `factorial`: ognuna ha il proprio frame che memorizza il valore del suo parametro n , oltre al valore che verrà restituito. Quando, prima o poi, termina la prima invocazione ricorsiva, restituisce $(n - 1)!$, valore che viene, poi, moltiplicato per n per calcolare $n!$, all'interno della prima invocazione del metodo `factorial`, quella che ha dato origine al processo ricorsivo.

La pila degli operandi

È interessante osservare che, in effetti, la JVM usa una pila anche per uno scopo completamente diverso. Le espressioni aritmetiche, come $((a + b) * (c + d))/e$, vengono valutate dalla JVM usando una *pila degli operandi* (*operand stack*). Una semplice operazione binaria (cioè con due operandi), come $a + b$, viene valutata inserendo a nella pila, poi inserendo b e, infine, invocando un'istruzione che estrae dalla pila due elementi, esegue la prevista operazione binaria usandoli come operandi e, poi, inserisce il risultato in cima alla pila. Analogamente, le istruzioni che scrivono dati in memoria o che leggono dati dalla memoria richiedono l'uso, rispettivamente, dei metodi `pop` e `push` applicati alla pila degli operandi. Quindi, la JVM usa una pila per valutare le espressioni aritmetiche in Java.

15.1.2 Riservare spazio nella memoria *heap*

Abbiamo già visto (nel Paragrafo 15.1.1) come la Java Virtual Machine memorizzi le variabili locali di un metodo all'interno del *frame* di quel metodo nella pila di esecuzione di Java, ma tale pila non è l'unica zona di memoria disponibile per i dati dei programmi in Java.

Assegnazione dinamica della memoria

La memoria occupata da un oggetto può anche essere assegnata all'oggetto stesso in modo dinamico durante l'esecuzione di un metodo, quando quel metodo utilizza lo speciale operatore `new` predefinito nel linguaggio Java. Ad esempio, il seguente enunciato, in Java, crea un array di numeri interi la cui dimensione è data dal valore della variabile `k`:

```
int[] items = new int[k];
```

La dimensione dell'array `items` è nota solamente al momento dell'esecuzione. Inoltre, l'array può continuare a esistere anche dopo che il metodo che l'ha creato ha terminato la propria esecuzione, quindi la memoria destinata a questo array non può essere una porzione della pila di esecuzione di Java.

La memoria *heap*

Invece di usare, per questo oggetto, la memoria della pila di Java, la JVM usa memoria che si trova in un'altra zona, la cosiddetta *memoria heap* (che non va confusa con la struttura dati "heap" che abbiamo presentato nel Capitolo 9). La Figura 15.3 mostra una rappresentazione schematica della memoria gestita dalla JVM, dove si può vedere la memoria *heap* e altre zone di memoria. Lo spazio disponibile all'interno della memoria *heap* è suddiviso

in *blocchi*, che sono “pezzi” (*chunk*) di memoria contigui, che possono avere dimensione variabile o fissa, e, in pratica, sono molto simili a un array.

Per rendere più semplice questa discussione, ipotizziamo che i blocchi della memoria heap abbiano dimensione fissa, diciamo 1024 byte, e che un blocco sia sufficientemente grande per qualsiasi tipo di oggetto che vogliamo creare, pur consapevoli del fatto che la gestione efficiente del caso più generale è effettivamente un interessante problema che merita ulteriori ricerche.

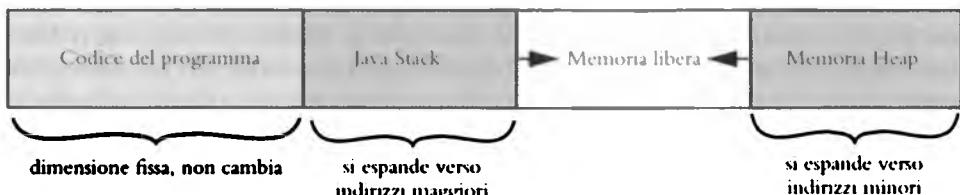


Figura 15.3: Una visione schematica della disposizione degli indirizzi di memoria all'interno della Java Virtual Machine. I valori degli indirizzi aumentano procedendo da sinistra a destra, come nella Figura 15.1.

Algoritmi per l'assegnazione della memoria

La definizione della Java Virtual Machine richiede che la memoria heap sia in grado di assegnare velocemente memoria ai nuovi oggetti, ma non specifica quale debba essere l'algoritmo con cui si raggiunge tale obiettivo. Un metodo piuttosto diffuso consiste nella memorizzazione all'interno di una lista concatenata (detta *free list*) dei riferimenti a zone di memoria libera disponibili (chiamate “buchi”, *hole*). I collegamenti che uniscono questi buchi vengono memorizzati all'interno dei buchi stessi, perché quella memoria non è in uso. Mentre la memoria viene assegnata a oggetti e, poi, resa libera quando gli oggetti non servono più, l'elenco dei buchi nella *free list* cambia, con la memoria inutilizzata che viene suddivisa in buchi disgiunti, divisi da blocchi di memoria in uso. Questa suddivisione della memoria libera in buchi disgiunti prende il nome di **frammentazione** (*fragmentation*): il problema che ne consegue è che diventa sempre più difficile trovare zone di memoria libera di grandi dimensioni, quando servono, anche se può esserci una quantità equivalente di memoria inutilizzata (ma frammentata).

Possono accadere due tipi di frammentazione. La **frammentazione interna** si verifica quando una porzione di un blocco di memoria che è stato assegnato rimane, in realtà, inutilizzata. Ad esempio, un programma potrebbe richiedere alla JVM di creare un array di dimensione 1000, usandone, però, soltanto le prime 100 celle. Un ambiente di esecuzione non può far molto per ridurre la frammentazione interna. La **frammentazione esterna**, d'altra parte, si verifica quando una quantità significativa di memoria non utilizzata si trova interposta a diversi blocchi contigui di memoria in uso. Dato che l'ambiente d'esecuzione ha il pieno controllo dell'assegnazione della memoria nel momento in cui questa viene richiesta (ad esempio, quando in Java viene usata la parola chiave `new`), dovrebbe usare una politica di assegnazione che cerchi in qualche modo di ridurre la frammentazione esterna.

Sono state proposte diverse strategie euristiche per assegnare la memoria dello heap in modo da minimizzare la frammentazione esterna. L'algoritmo *best-fit* (“corrispondenza

migliore") scandisce l'intera *free list* per trovare il buco la cui dimensione è più vicina alla quantità di memoria richiesta. L'algoritmo *first-fit* ("il primo che va bene") scandisce la *free list* dall'inizio e sceglie il primo buco che sia grande abbastanza. L'algoritmo *next-fit* ("il prossimo che va bene") è simile, perché scandisce la *free list* e sceglie il primo buco che sia grande abbastanza, ma, invece di partire sempre dall'inizio della lista, comincia la ricerca a partire dal punto in cui era arrivato durante la ricerca precedente, considerando la lista come se fosse circolare (si veda il Paragrafo 3.3). L'algoritmo *worst-fit* ("corrispondenza peggiore") cerca nella *free list* il buco più grande che c'è, operazione che potrebbe risultare più veloce di una scansione dell'intera lista se tale lista viene gestita come coda prioritaria (si veda il Capitolo 9). In ogni algoritmo, la quantità di memoria richiesta viene sottratta da quella libera nel buco scelto e la parte rimanente del buco resta nella *free list*.

Anche se a un primo esame l'algoritmo *best-fit* potrebbe sembrare il migliore, in realtà tende a produrre la peggiore frammentazione esterna, perché le parti restanti dei buchi scelti tendono a essere piccole. L'algoritmo *first-fit* è veloce, ma tende a produrre molta frammentazione esterna nella parte iniziale della *free list*, fenomeno che rallenta le ricerche successive, che partono sempre dall'inizio. L'algoritmo *next-fit* distribuisce la frammentazione in modo più equo all'interno della memoria heap, tenendo in questo modo basso il tempo di ricerca; questa distribuzione rende, però, più difficile la ricerca di blocchi di grande dimensione. L'algoritmo *worst-fit* cerca di evitare questo problema facendo in modo che le porzioni di memoria libera abbiano le massime dimensioni possibili.

15.1.3 Garbage collection

In alcuni linguaggi di programmazione, come C e C++, lo spazio di memoria occupato dagli oggetti deve essere liberato in modo esplicito dal programmatore quando gli oggetti non servono più: un'attività spesso ignorata dai programmatore principianti, all'origine di frustranti errori di programmazione anche per i programmatore più esperti. I progettisti del linguaggio Java hanno affidato completamente all'ambiente d'esecuzione l'onere della gestione della memoria.

Come già detto, la memoria occupata dagli oggetti viene assegnata all'interno della memoria heap, mentre lo spazio destinato alle variabili locali di un programma Java in esecuzione si trova all'interno della pila dei metodi (i programmi semplici di cui parliamo in questo libro vengono tipicamente eseguiti in un unico thread, quindi hanno un'unica pila dei metodi, ma, in generale, c'è una pila dei metodi per ogni thread in esecuzione). Dato che le variabili presenti nella pila dei metodi possono fare riferimento a oggetti che si trovano nella memoria heap, chiamiamo *oggetti radice* (*root object*) tutte le variabili e tutti gli oggetti che si trovano nella pila dei metodi di qualsiasi thread in esecuzione, mentre tutti gli oggetti che possono essere raggiunti seguendo un riferimento che parte da un *oggetto radice* sono detti *oggetti vivi* (*live object*). Gli *oggetti vivi* sono gli oggetti attivi, attualmente utilizzati dal programma in esecuzione, e la memoria che occupano *non* deve essere resa libera. Ad esempio, un programma Java può memorizzare in una variabile un riferimento a una sequenza *S*, implementata mediante una lista doppiamente concatenata. La variabile che fa riferimento a *S* è un oggetto radice, mentre *S* è un oggetto vivo, così come sono vivi tutti gli oggetti di tipo "nodo" che sono raggiungibili, direttamente o indirettamente, da *S* e tutti gli elementi a cui fanno riferimento i nodi.

Di quando in quando, la Java Virtual Machine può accorgersi che lo spazio disponibile all'interno della memoria heap si sta esaurendo: in quel momento, la JVM può decidere di rendere libero lo spazio utilizzato da oggetti che non sono più vivi, restituendo tale memoria alla *free list*. Questa procedura di liberazione dello spazio di memoria occupato inutilmente si chiama *garbage collection* ("raccolta della spazzatura"). Esistono diversi algoritmi che realizzano la *garbage collection*, ma uno dei più usati è l'algoritmo *mark-sweep*.

L'algoritmo *mark-sweep*

Con l'algoritmo *mark-sweep* per la *garbage collection*, usiamo un bit (detto "mark") per contrassegnare ciascun oggetto, dicendo così se l'oggetto è vivo oppure no. Quando, a un certo punto, la JVM decide che c'è bisogno di "raccogliere la spazzatura", vengono sospese tutte le altre attività e vengono azzerati tutti i bit di marcatura di tutti gli oggetti presenti in quel momento all'interno della memoria heap. Poi, si scandisce la pila dei metodi dei thread in esecuzione e si contrasseggiano come "vivi" tutti gli oggetti radice. Infine, bisogna individuare tutti gli altri oggetti vivi, quelli che sono in qualche modo raggiungibili a partire dagli oggetti radice.

Per fare questo in modo efficiente, possiamo effettuare un attraversamento in profondità (descritto nel Paragrafo 14.3.1) sul grafo orientato definito dagli oggetti che fanno riferimento ad altri oggetti. In questo caso, ogni oggetto della memoria heap è visto come un vertice di un grafo orientato, mentre i riferimenti da un oggetto a un altro sono i lati orientati di tale grafo. Eseguendo un DFS orientato a partire da ciascun oggetto radice, possiamo individuare correttamente e contrassegnare tutti gli oggetti vivi. Questa procedura è la fase "mark".

Terminata questa procedura, si effettua una scansione della memoria heap e si rende libero lo spazio utilizzato dagli oggetti che non sono stati contrassegnati come vivi. Allo stesso tempo, possiamo anche opzionalmente fondere insieme tutto lo spazio occupato nella memoria heap in modo da costituire un unico blocco, eliminando così per il momento la frammentazione esterna. Questa procedura di scansione e recupero della memoria è la fase "sweep" (cioè "fare pulizia") e, quando è terminata, si può riprendere l'esecuzione del programma, che era stata sospesa. In questo modo, la *garbage collection* mediante *mark-sweep* è in grado di recuperare lo spazio di memoria inutilizzato in un tempo proporzionale al numero di oggetti vivi e dei loro riferimenti, sommato alla dimensione della memoria heap.

Eseguire DFS sul posto

L'algoritmo *mark-sweep* recupera correttamente lo spazio inutilizzato all'interno della memoria heap, ma durante la fase *mark* ci dobbiamo confrontare con un aspetto importante. Dato che stiamo recuperando spazio di memoria in un momento in cui lo spazio disponibile è scarso, dobbiamo fare attenzione a non usare spazio aggiuntivo durante l'operazione stessa di *garbage collection*. Il problema è che l'algoritmo DFS, nell'implementazione ricorsiva che abbiamo descritto nel Paragrafo 14.3.1, può utilizzare uno spazio proporzionale al numero di vertici del grafo. In questa situazione, i vertici del nostro grafo sono gli oggetti presenti nella memoria heap, quindi probabilmente non abbiamo così tanta memoria disponibile. L'alternativa è quella di trovare un modo per eseguire DFS sul posto, anziché ricorsivamente.

L'idea principale per eseguire DFS sul posto è quella di simulare la pila di ricorsione usando i lati del grafo (che, nel caso della *garbage collection*, corrispondono ai riferimenti agli oggetti). Quando percorriamo un lato per andare da un vertice visitato v a un nuovo vertice

w , modifichiamo il lato (v, w) memorizzato nella lista di adiacenze di v in modo che punti al genitore di v nell'albero DFS. Quanto, poi, torniamo nel nodo v (e dobbiamo simulare il fatto che sia terminata l'invocazione ricorsiva fatta partire in w), possiamo ripristinare il lato che avevamo modificato in modo che punti nuovamente verso w . Ovviamente ci serve un modo per identificare i lati che devono essere ripristinati. Una possibilità è quella di numerare i riferimenti uscenti da v come 1, 2, e così via, memorizzando, oltre al bit che dice se l'oggetto è vivo (che, durante l'attraversamento DFS, usiamo come indicatore di vertice "visitato"), un identificatore a contatore che dica quali lati sono stati modificati.

L'uso di un contatore richiede un'ulteriore parola di memoria per ogni oggetto, che, però, in alcune implementazioni può essere evitata. Ad esempio, molte implementazioni della JVM rappresentano un oggetto come una composizione di un riferimento a un identificatore di tipo (che indica se l'oggetto è, ad esempio, un esemplare di Integer o di qualche altra classe) e un riferimento agli oggetti o campi contenuti nell'oggetto. Dato che, in tali implementazioni, il riferimento al tipo è sempre il primo elemento dell'oggetto composito, possiamo usarlo per "contrassegnare" (*mark*) il lato che abbiamo modificato uscendo dall'oggetto v e andando in qualche altro oggetto w : semplicemente, scambiamo in v il riferimento che punta al tipo di v con il riferimento che punta a w . Quando, poi, ritorniamo in v , possiamo rapidamente identificare il lato (v, w) che abbiamo modificato, perché sarà il primo riferimento nell'oggetto composito corrispondente a v , e la posizione del riferimento al tipo di v ci dirà la posizione a cui appartiene il lato nella lista di adiacenze di v .

15.2 Gerarchie di memoria e *caching*

Con il crescente utilizzo dei calcolatori nella società, le applicazioni software devono gestire quantità di dati di enormi dimensioni. Come esempi di tali applicazioni possiamo citare l'elaborazione di transazioni finanziarie *online*, l'organizzazione e la gestione di basi di dati e l'analisi della storia e delle preferenze di acquisto della clientela. La quantità di dati può essere così grande che a volte le prestazioni complessive degli algoritmi e delle strutture dati dipendono più dal tempo di accesso ai dati che dalla velocità di elaborazione della CPU.

15.2.1 Sistemi di memoria

Per poter elaborare grandi insiemi di dati, i computer utilizzano una *gerarchia* di diversi tipi di memorie, che si distinguono per dimensione e per distanza dalla CPU. I registri interni alla CPU, utilizzati dalla CPU stessa, rappresentano il tipo di memoria più "vicino" alla CPU: l'accesso a tale memoria è estremamente veloce, ma le posizioni ("parole") disponibili al suo interno sono veramente poche. Al secondo livello della gerarchia troviamo una o più memorie *cache* (il significato della parola è "nascondiglio segreto"): sono significativamente più capienti dell'insieme di registri interni di una CPU, ma il tempo d'accesso è maggiore. Al terzo livello della gerarchia si pone la *memoria interna*, chiamata anche *memoria principale*: molto più capiente della memoria cache, con tempo d'accesso ancora maggiore. Il livello successivo è la *memoria esterna*, solitamente costituita da dischi, CD, DVD e/o nastri magnetici: una memoria di grandi dimensioni, ma anche molto lenta. I dati memorizzati in una rete esterna possono essere considerati come un ulteriore livello di questa gerarchia,

con una capacità e un tempo d'accesso ancora maggiori. Quindi, la gerarchia di memoria dei computer può essere vista come costituita da cinque o più livelli, ciascuno dei quali è più capiente e più lento del livello precedente, come si può vedere nella Figura 15.4. Durante l'esecuzione di un programma, i dati vengono continuamente copiati da un livello della gerarchia a uno dei livelli adiacenti, e questi trasferimenti possono diventare un collo di bottiglia computazionale.

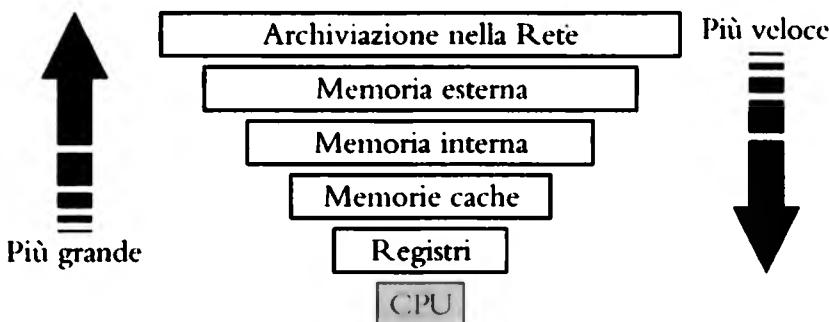


Figura 15.4: La gerarchia della memoria.

15.2.2 Strategie di gestione della memoria cache

L'influenza della gerarchia di memoria sulle prestazioni di un programma dipende fortemente dalla dimensione del problema che si cerca di risolvere e dalle caratteristiche fisiche del sistema di calcolo. Spesso il collo di bottiglia è posto tra due livelli della gerarchia di memoria, il livello che è in grado di memorizzare tutti i dati del problema e il livello immediatamente sottostante a quello. Per un problema che può essere completamente memorizzato nella memoria principale, i due livelli più importanti sono la memoria cache e la memoria interna. I tempi d'accesso caratteristici della memoria interna possono essere anche 10 o 100 volte maggiori dei tempi tipici delle memorie cache, quindi è preferibile riuscire a effettuare la maggior parte degli accessi ai dati quando questi si trovano nella memoria cache. Se, d'altra parte, il problema non può risiedere completamente nella memoria principale, i due livelli più importanti della gerarchia di memoria sono la memoria interna e la memoria esterna. In questo caso le differenze sono ancora più rilevanti, perché i tempi d'accesso ai dati memorizzati su un disco, il dispositivo di memoria esterna più diffuso, sono tipicamente 100 mila o un milione di volte maggiori di quelli tipici della memoria interna.

Per farvi un'idea più precisa del significato di questi numeri, immaginate che ci sia uno studente a Baltimora che vuole inviare una richiesta di denaro ai suoi genitori che vivono a Chicago. Se lo studente invia la richiesta con un messaggio di posta elettronica, può arrivare a destinazione in circa cinque secondi. Pensate a questa modalità di comunicazione come se fosse corrispondente all'accesso alla memoria interna da parte della CPU. La modalità di comunicazione corrispondente all'accesso a una memoria esterna che sia 500 mila volte più lenta della memoria interna sarebbe equivalente al fatto che lo studente si recasse di persona a Chicago a consegnare il messaggio, a piedi, cosa che richiederebbe circa un mese di tempo se camminasse per 20 miglia al giorno. Di conseguenza, dobbiamo cercare di fare il minor numero di accessi possibile alla memoria esterna.

La maggior parte degli algoritmi non sono stati progettati tenendo presente la gerarchia di memoria, nonostante la grande variabilità tra i tempi di accesso caratteristici dei diversi livelli. In effetti, in tutte le analisi di algoritmi condotte fin qui in questo libro abbiamo sempre ipotizzato che tutti gli accessi alla memoria richiedessero lo stesso tempo. Questa ipotesi può sembrare, a prima vista, un grave errore, oltretutto un errore di cui ci stiamo accorgendo solamente nel capitolo conclusivo, ma ci sono valide ragioni per ritenere ragionevole una tale ipotesi.

Una giustificazione per questa ipotesi sta nel fatto che spesso è necessario ipotizzare che tutti gli accessi alla memoria richiedano lo stesso tempo, magari semplicemente perché non è facile disporre di informazioni relative alle dimensioni delle memorie poste ai vari livelli della gerarchia. Ad esempio, un programma Java che sia stato progettato per poter essere eseguito su molte diverse piattaforme di calcolo non potrebbe essere agevolmente caratterizzato nei confronti di una specifica configurazione dell'architettura di un computer. Se ne disponiamo, possiamo certamente utilizzare informazioni specifiche di una determinata architettura (e più avanti in questo stesso capitolo mostreremo come si possano sfruttare queste informazioni), ma, dopo aver ottimizzato il nostro software per una specifica configurazione del calcolatore, il programma non sarebbe più indipendente dalla piattaforma di esecuzione. Fortunatamente, non sempre tali ottimizzazioni sono necessarie, principalmente a causa della seconda motivazione che portiamo a favore dell'ipotesi di tempi d'accesso equalizzati.

Strategie di caching e blocking

Un'altra valida motivazione per ritenere uguali i tempi d'accesso alla memoria durante l'esecuzione di un algoritmo è conseguenza del fatto che i progettisti di sistemi operativi hanno sviluppato dei meccanismi di applicabilità generale che consentono di avere tempi d'accesso veloci nella maggior parte dei casi. Queste strategie si basano su due importanti proprietà di *località dei riferimenti*, che caratterizzano la maggior parte degli algoritmi:

- **Località temporale:** Se un programma accede a una determinata posizione nella memoria, aumenta la probabilità che presto acceda di nuovo alla stessa posizione. Ad esempio, spesso si usa il valore di una variabile contatore in diverse espressioni, tra le quali l'incremento del valore del contatore. In effetti, una massima molto ripetuta dai progettisti architettonici di calcolatori dice che un programma trascorre il 90% del suo tempo eseguendo il 10% del suo codice.
- **Località spaziale:** Se un programma accede a una determinata posizione nella memoria, aumenta la probabilità che presto acceda a posizioni vicine a quella. Ad esempio, un programma che usa un array accederà con molta probabilità alle posizioni dell'array in ordine sequenziale o quasi sequenziale.

I ricercatori e gli ingegneri del settore informatico hanno eseguito molti e approfonditi esperimenti di analisi del profilo di comportamento del software, dimostrando la correttezza dell'affermazione secondo la quale la maggior parte del software è caratterizzato da entrambe queste tipologie di località dei riferimenti. Ad esempio, un ciclo annidato a contatore che venga usato per scandire ripetutamente un array manifesterà entrambe queste tipologie di località.

Queste due proprietà di località, temporale e spaziale, hanno dato origine a due scelte progettuali fondamentali per i sistemi di memoria multilivello, che sono solitamente implementate nell'interfaccia tra la memoria cache e la memoria interna, e anche nell'interfaccia tra la memoria interna e la memoria esterna.

La prima scelta progettuale è chiamata *memoria virtuale (virtual memory)* e coinvolge due livelli consecutivi di memoria, che chiamiamo primario e secondario e che solitamente sono, rispettivamente, la memoria cache e la memoria interna, ma possono essere anche, rispettivamente, la memoria interna e la memoria esterna. Questo concetto consiste nel mettere a disposizione dei programmi uno spazio di indirizzamento della memoria che abbia la stessa capacità della memoria del livello secondario, trasferendo i dati dal livello secondario al livello primario quando questi vengono effettivamente utilizzati (in base all'indirizzo). La memoria virtuale non vincola il programmatore a rimanere all'interno della dimensione effettiva della memoria primaria, ma può lavorare sfruttando la maggiore dimensione della memoria secondaria. Il trasferimento "su richiesta" dei dati nella memoria primaria è chiamato *caching* e trova la sua motivazione nella località temporale: copiando dati nella memoria primaria, si spera che presto verranno utilizzati e, in questo modo, il sistema di memoria sarà in grado di rispondere velocemente a tutte le richieste relative a questi dati che si verificheranno nell'immediato futuro.

La seconda scelta progettuale trova, invece, motivazione nella località spaziale. Nello specifico, se viene effettuato un accesso all'indirizzo l nella memoria secondaria, viene portato nella memoria primaria un grande blocco di parole contigue che contenga l'indirizzo l (si veda la Figura 15.5). Questa strategia prende il nome di *blocking* ("gestione a blocchi") ed è motivata dalla elevata probabilità che, per la località spaziale, venga presto richiesto l'accesso a indirizzi vicini a l . Nell'interfaccia tra la memoria cache e la memoria interna, questi blocchi sono spesso chiamati *righe di cache (cache line)*, mentre all'interfaccia tra la memoria interna e la memoria esterna si chiamano *pagine (page)*.

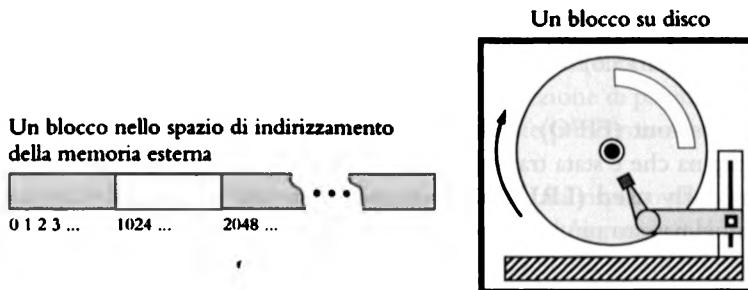


Figura 15.5: Blocchi nella memoria esterna.

Implementando la memoria virtuale con le strategie di *caching* e *blocking*, spesso si ha la sensazione che la memoria secondaria sia più veloce di quello che è in realtà, ma c'è ancora un problema: la memoria primaria ha una capacità molto inferiore di quella secondaria. Inoltre, dal momento che i sistemi di memoria usano la strategia di *blocking*, qualsiasi programma sufficientemente impegnativo arriverà prima o poi al punto in cui richiederà dati che si trovano nella memoria secondaria, ma la memoria primaria è già piena di blocchi richiesti

in precedenza. Per poter soddisfare la richiesta e continuare a utilizzare le strategie di *caching* e *blocking*, il sistema deve eliminare un blocco dalla memoria primaria, per far spazio al nuovo blocco che è stato richiesto e che deve essere trasferito dalla memoria secondaria. La strategia di decisione in merito a quale debba essere il blocco espulso genera un certo numero di problemi interessanti, che coinvolgono algoritmi e strutture dati.

Caching nei browser web

Per motivarci meglio, considereremo un problema simile, che si incontra quando si torna a consultare informazioni contenute in pagine web già visitate in precedenza. Per sfruttare la località temporale dei riferimenti, spesso è vantaggioso memorizzare copie di pagine web in una memoria *cache*, in modo da recuperarle velocemente quando verranno richieste di nuovo. Questa strategia crea a tutti gli effetti una gerarchia di memoria a due livelli per il browser, con la cache che svolge la funzione di piccola e veloce memoria interna e la rete che rappresenta la memoria esterna. In particolare, supponiamo che la memoria cache disponga di m posizioni in cui memorizzare pagine web e ipotizziamo che una pagina web possa essere inserita in qualunque posizione libera: in questo caso si parla di cache *completamente associativa* (*fully associative*).

Il browser web, durante la propria esecuzione, riceve richieste per diverse pagine web. Ogni volta che viene richiesta la pagina p , il browser determina (con una verifica veloce presso il server) se p è rimasta immutata ed è ancora contenuta nella cache. Se p è contenuta nella cache, allora il browser soddisfa la richiesta usando tale copia. Se, invece, p non si trova nella cache, la pagina p viene richiesta in Internet e il browser la copia all'interno della cache: se una delle m posizioni della cache è disponibile, allora vi copia la pagina p , ma se tutte le m posizioni sono occupate, il browser deve determinare quale pagina possa essere eliminata dalla cache prima di potervi copiare p . Anche in questo caso, ovviamente, si possono adottare molte strategie diverse per prendere questa decisione.

Algoritmi di sostituzione delle pagine

Ecco alcuni dei più noti algoritmi di sostituzione delle pagine nelle memorie cache (si veda anche la Figura 15.6):

- **First-in, first-out (FIFO):** Elimina la pagina che si trova nella cache da più tempo, cioè la pagina che è stata trasferita nella cache nel passato più lontano.
- **Least recently used (LRU, usata meno recentemente):** Elimina la pagina che è stata richiesta nel passato più lontano.

Inoltre, prendiamo in esame anche una strategia molto semplice, puramente casuale:

- **Random:** Elimina una pagina scelta a caso.

La strategia casuale è una di quelle più facili da implementare, perché richiede solamente un generatore di numeri casuali o pseudocasuali: per ogni pagina sostituita, è richiesto soltanto un tempo $O(1)$ oltre al tempo effettivamente usato per sostituire la pagina. Inoltre, per ogni pagina richiesta, oltre al fatto di dover determinare se è presente nella cache oppure no, non si impiega altro tempo, anche se, ovviamente, questa strategia non fa nessun tentativo di sfruttare a proprio vantaggio un'eventuale località temporale che caratterizzi l'attività di navigazione in Internet dell'utente.

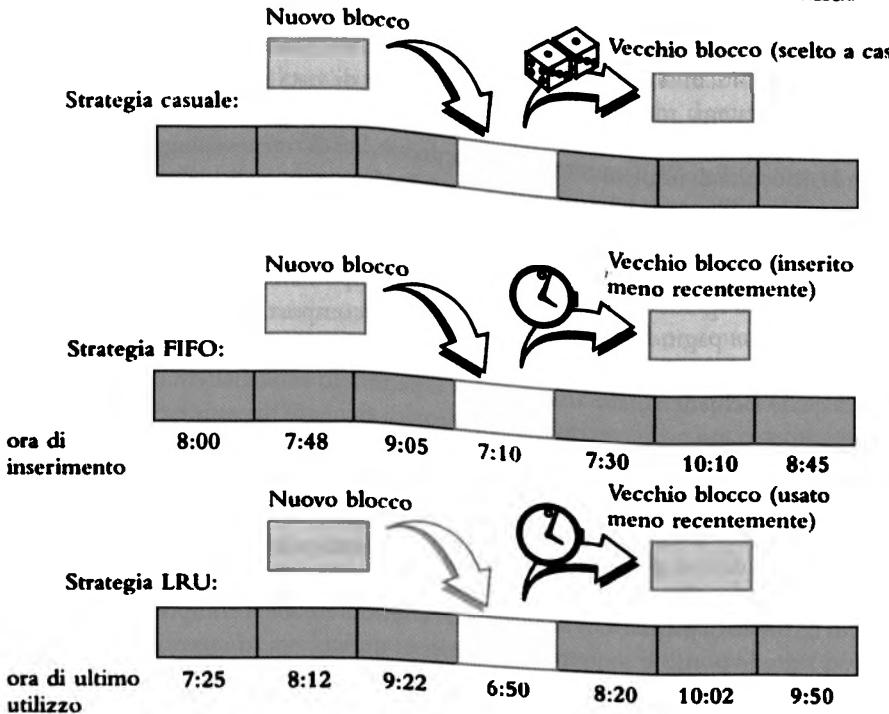


Figura 15.6: Le strategie di sostituzione casuale, FIFO e LRU.

La strategia FIFO è forse altrettanto semplice da implementare, dato che richiede solamente una coda Q per memorizzare i riferimenti alle pagine presenti nella cache. Le pagine vengono accodate in Q nel momento in cui vengono inserite nella cache dal browser: quando c'è bisogno di eliminare una pagina dalla cache, il browser esegue semplicemente un'operazione di estrazione da Q , determinando così la pagina da eliminare. Questa strategia, quindi, richiede un tempo aggiuntivo $O(1)$ per ogni sostituzione di pagina e non necessita di elaborazione aggiuntiva durante la gestione delle richieste, pur cercando di trarre qualche vantaggio dalla località temporale.

La strategia LRU rappresenta un ulteriore passo in avanti rispetto alla gestione FIFO, perché sfrutta esplicitamente a proprio vantaggio la località temporale nel miglior modo possibile, eliminando sempre la pagina che è stata utilizzata meno recentemente. Dal punto di vista della strategia stessa, si tratta di una scelta eccellente, ma la sua implementazione è onerosa: il modo in cui sfrutta la località temporale e spaziale è abbastanza costoso. Per implementare la strategia LRU serve una coda prioritaria modificabile, Q , che consenta l'aggiornamento della chiave associata alla pagina (che è l'istante di tempo dell'ultimo accesso alla pagina). Se Q viene implementata con una sequenza ordinata che usi come supporto una lista concatenata, allora la decisione sulla pagina da eliminare può essere presa in un tempo $O(1)$, mentre quando si inserisce una nuova pagina in Q o se ne modifica la chiave (perché viene effettuato un nuovo accesso alla pagina), alla pagina viene sempre assegnata la chiave più alta tra quelle presenti in Q (l'istante attuale...) e, quindi, viene posizionata in fondo alla lista, operazione che, di nuovo, richiede un tempo $O(1)$. Anche se la strategia

LRU, implementata nel modo appena descritto, richiede tempi addizionali costanti, i fattori costanti coinvolti, in termini di tempo e di spazio necessario per la coda prioritaria Q, rendono questa strategia meno attrattiva da un punto di vista pratico.

Dato che queste diverse strategie di sostituzione delle pagine prevedono compromessi diversi tra la difficoltà di implementazione e la possibilità di trarre vantaggio dalla località, è naturale porsi il problema di effettuare una qualche forma di analisi comparativa tra questi metodi, per vedere se ne esiste uno che sia migliore degli altri.

Da un punto di vista del caso peggiore, le strategie FIFO e LRU hanno un comportamento abbastanza sgradevole, che le accomuna. Supponiamo, ad esempio, di avere una cache contenente m pagine e prendiamo in esame il comportamento dei metodi FIFO e LRU per la sostituzione delle pagine quando viene eseguito un programma che contiene un ciclo, il quale richiede ripetutamente $m + 1$ pagine in modalità circolare. Entrambe le strategie si comportano male con questa sequenza di richieste di pagine, perché richiedono una sostituzione di pagina in risposta a ogni singola richiesta. Quindi, nel caso peggiore, queste strategie sono, in pratica, le peggiori che si possono immaginare: necessitano di una sostituzione di pagina a ogni richiesta.

Questa analisi di caso peggiore, però, è un po' troppo pessimistica, perché si concentra sul comportamento assunto da ciascuna di queste strategie quando si verifica una pessima sequenza di richieste di pagine. Un'analisi ideale confronterebbe il comportamento di questi metodi con tutte le possibili sequenze di richieste: questo, ovviamente, non si può fare in modo esaustivo, ma sono state fatte numerosissime simulazioni sperimentali su sequenze di richieste di pagine effettivamente derivate dal comportamento di programmi reali. Sulla base di questi confronti sperimentali, la strategia LRU si è rivelata superiore rispetto alla strategia FIFO, la quale, a sua volta, è solitamente migliore della strategia casuale.

15.3 Ricerche esterne e B -alberi

Consideriamo il problema di gestire una grande raccolta di oggetti che non trova posto nella memoria principale, come avviene nel caso di una tipica base di dati (*database*). In questo contesto, i blocchi della memoria secondaria sono *blocchi del disco* (*disk block*) e, analogamente, il trasferimento di un blocco tra la memoria secondaria e la memoria primaria viene detto *trasferimento dal disco* (*disk transfer*). Ricordando l'enorme differenza che esiste tra i tempi d'accesso alla memoria interna e alla memoria esterna (cioè il disco), l'obiettivo principale della gestione di questi dati nella memoria esterna è quello di minimizzare il numero di trasferimenti dal disco che sono necessari per eseguire un'interrogazione della base di dati (o un suo aggiornamento). Questo conteggio viene spesso chiamato *complessità di I/O* (che sta per *Input/Output*) dell'algoritmo in esame.

Alcune rappresentazioni inefficienti nella memoria esterna

Una tipica operazione che vogliamo consentire è la ricerca in una mappa mediante una chiave. Se memorizzassimo n oggetti non ordinati in una catena doppiamente concatenata, la ricerca di una specifica chiave all'interno della lista richiederebbe n trasferimenti, nel caso peggiore, perché ogni collegamento seguito nella lista concatenata, per passare da un nodo a un altro, potrebbe accedere a un blocco di memoria diverso.

Possiamo ridurre il numero di blocchi trasferiti memorizzando la sequenza in un array. Sfruttando la località spaziale dei riferimenti, una ricerca sequenziale in un array può essere eseguita trasferendo soltanto un numero $O(n/B)$ di blocchi, dove B indica il numero di elementi dell'array che trovano posto in un blocco: infatti, il trasferimento del primo blocco, che avviene in seguito all'accesso al primo elemento dell'array, recupera, in effetti, i primi B elementi dell'array stesso, e così via per ciascun successivo blocco. È bene notare che il limite di $O(n/B)$ trasferimenti si raggiunge, in Java, soltanto quando si usa un array di valori di tipo primitivo. Per un array di oggetti, l'array memorizza una sequenza di riferimenti e gli oggetti a cui questi fanno riferimento non sono necessariamente posizionati uno accanto all'altro in memoria, per cui, nel caso peggiore, ci possono comunque essere n blocchi trasferiti.

Se in un array viene memorizzata una sequenza *ordinata*, l'esecuzione di una ricerca binaria richiede l'accesso a $O(\log_2 n)$ valori, che è un bel miglioramento rispetto alla ricerca sequenziale, ma, passando al numero di blocchi trasferiti, non si ha nessun ulteriore vantaggio significativo, perché è probabile che ogni richiesta di accesso durante la ricerca binaria sia riferita a un blocco diverso. Inoltre, le operazioni di aggiornamento di un array ordinato sono, come al solito, onerose.

Dato che queste semplici implementazioni sono inefficienti dal punto di vista dell'I/O, passiamo a considerare le strategie che usano alberi binari bilanciati (come gli alberi AVL o gli alberi rosso-nero) che, con riferimento alla sola memoria interna, richiedono tempi logaritmici, così come altre strutture che sono logaritmiche nel caso medio (come, ad esempio, *skip list* e alberi *splay*). Tipicamente, ogni accesso a un nodo di queste strutture, durante una ricerca o una modifica, farà riferimento a un blocco diverso, quindi tutti questi metodi richiedono un numero di trasferimenti $O(\log_2 n)$ nel caso peggiore, per eseguire una ricerca o una modifica. Ma possiamo fare meglio! Possiamo eseguire ricerche e aggiornamenti usando soltanto $O(\log_B n) = O(\log n / \log B)$ trasferimenti.

15.3.1 Alberi (a, b)

Per ridurre il numero di accessi alla memoria esterna durante una ricerca, possiamo rappresentare la nostra mappa usando un albero di ricerca a più vie (presentato nel Paragrafo 11.5.1). Questo approccio porta a una generalizzazione della struttura dati che abbiamo chiamato “albero $(2, 4)$ ”, che diventa l’albero (a, b) .

Un albero (a, b) è un albero di ricerca a più vie in cui ogni nodo ha un numero di figli compreso tra a e b e memorizza un numero di voci compreso tra $a - 1$ e $b - 1$. Gli algoritmi usati per fare ricerche, inserimenti e rimozioni in un albero (a, b) sono banali generalizzazioni dei corrispondenti algoritmi visti per gli alberi $(2, 4)$. Il vantaggio della generalizzazione, passando da alberi $(2, 4)$ ad alberi (a, b) , sta nel fatto che una classe di alberi parametrici consente una struttura di ricerca flessibile, dove la dimensione dei nodi e il tempo d'esecuzione delle diverse operazioni agenti sulla mappa dipendono dai parametri a e b : impostandoli a valori adeguati, in relazione alla dimensione dei blocchi sul disco, possiamo ottenere una struttura dati che raggiunge buone prestazioni anche operando sulla memoria esterna.

Definizione di albero (a, b)

Un albero (a, b) , dove i parametri a e b sono numeri interi tali che $2 \leq a \leq (b + 1)/2$, è un albero di ricerca a più vie, T , con i seguenti ulteriori vincoli:

Proprietà della dimensione: Ogni nodo interno ha almeno a figli, a meno che non sia la radice, e ha al massimo b figli.

Proprietà della profondità: Tutti i nodi esterni hanno la stessa profondità.

Proposizione 15.1: L'altezza di albero (a, b) che contiene n voci è $\Omega(\log n/\log b)$ e $O(\log n/\log a)$.

Dimostrazione: Sia T un albero (a, b) che contiene n voci e sia h la sua altezza. Dimostreremo l'affermazione stabilendo, per h , la validità dei seguenti limiti:

$$\frac{1}{\log b} \log(n+1) \leq h \leq \frac{1}{\log a} \log \frac{n+1}{2} + 1.$$

Per le proprietà sulla dimensione e sulla profondità che fanno parte della definizione degli alberi (a, b) , il numero n'' di nodi esterni di T è almeno $2a^{h-1}$ e al massimo b^h . Inoltre, per la Proposizione 11.6, $n'' = n + 1$. Quindi:

$$2a^{h-1} \leq n + 1 \leq b^h.$$

Prendendo il logaritmo in base 2 di ciascun termine, otteniamo:

$$(h-1) \log a + 1 \leq \log(n+1) \leq h \log b.$$

e una semplice manipolazione algebrica di queste diseguaglianze completa la dimostrazione. ■

Operazioni di ricerca e aggiornamento

Ricordiamo, dal Paragrafo 11.5.1, che in un albero di ricerca a più vie, T , ciascun nodo w contiene una struttura secondaria $M(w)$, che è a sua volta una mappa. Se T è un albero (a, b) , allora $M(w)$ contiene al massimo b voci. Indichiamo con $f(b)$ il tempo richiesto per eseguire una ricerca in una delle mappe $M(w)$. L'algoritmo di ricerca in un albero (a, b) è esattamente uguale a quello che funziona in qualsiasi albero di ricerca a più vie e riportato nel Paragrafo 11.5.1. Quindi, la ricerca in un albero (a, b) , T , contenente n voci richiede un tempo $O\left(\frac{f(b)}{\log a} \log n\right)$. Osserviamo che, se b è un valore che può essere considerato costante (e lo è anche a), allora il tempo di ricerca è $O(\log n)$.

La principale applicazione degli alberi (a, b) riguarda le mappe memorizzate nella memoria esterna. In particolare, per minimizzare il numero di accessi al disco, si selezionano i parametri a e b in modo che ogni nodo dell'albero trovi posto in un solo blocco del disco, così che, ipotizzando che soltanto i trasferimenti di blocchi diano un contributo al tempo d'esecuzione, sia $f(b) = 1$. Assegnando ai parametri a e b i valori corretti in questa situazione, si genera una struttura dati che prende il nome *B-albero*, che ora descriveremo brevemente. Prima di questo, però, parliamo di come si gestiscano negli alberi (a, b) gli inserimenti e le rimozioni.

L'algoritmo di inserimento in un albero (a, b) è simile a quello visto per gli alberi $(2, 4)$. Si verifica un *overflow* quando una voce viene inserita in un b -nodo, w , che diventa un $(b + 1)$ -nodo, che non è valido (ricordiamo che un nodo, in un albero a più vie, è un d -nodo se ha d figli). Per porre rimedio a una situazione di overflow, dividiamo il nodo w spostando la sua voce mediana nel genitore di w e sostituendo w con un $\lceil (b + 1)/2 \rceil$ -nodo, che chiamiamo w' , e un $\lfloor (b + 1)/2 \rfloor$ -nodo, che chiamiamo w'' , facendo così risultare evidente il motivo per cui, nella definizione di albero (a, b) , abbiamo richiesto che fosse $a \leq (b + 1)/2$. Osserviamo che, come conseguenza della divisione, sarà necessario costruire le strutture secondarie $M(w')$ e $M(w'')$.

L'eliminazione di una voce da un albero (a, b) è, di nuovo, simile all'omologa operazione eseguita in un albero $(2, 4)$. Si verifica una condizione di *underflow* quando viene eliminata una chiave da un a -nodo, w , diverso dalla radice, perché questo genera un $(a - 1)$ -nodo, che non è valido. Per porre rimedio a una situazione di underflow, effettuiamo un trasferimento che coinvolga un fratello di w che non sia un a -nodo, oppure una fusione di w con un fratello che sia un a -nodo. Il nuovo nodo w' risultante dalla fusione è un $(2a - 1)$ -nodo e questo è un altro motivo per imporre che sia $a \leq (b + 1)/2$.

La Tabella 15.1 mostra le prestazioni di una mappa realizzata con un albero (a, b) .

Tabella 15.1: Tempi d'esecuzione nel caso peggiore per le operazioni di una mappa realizzata con un albero (a, b) , T , nell'ipotesi che la struttura secondaria presente nei nodi di T consenta di effettuare ricerche in un tempo $f(b)$ e operazioni di divisione e fusione in un tempo $g(b)$, con le funzioni $f(b)$ e $g(b)$ che possono essere rese $O(1)$ se si contano solamente i trasferimenti da disco.

Metodo	Tempo d'esecuzione
get	$O\left(\frac{f(b)}{\log_a} \log n\right)$
put	$O\left(\frac{f(b)}{\log_a} \log n\right)$
remove	$O\left(\frac{f(b)}{\log_a} \log n\right)$

15.3.2 B-alberi

Una versione della struttura dati “albero (a, b) ”, che è il modo migliore, tra quelli noti, per gestire una mappa nella memoria esterna, è chiamata “B-albero”, di cui è visibile un esempio nella Figura 15.7. Un *B-albero di ordine d* è un albero (a, b) con $a = \lceil d/2 \rceil$ e $b = d$. Avendo già analizzato i metodi standard per eseguire ricerche e modifiche in un albero (a, b) , ci concentriamo qui sulla complessità di I/O dei B-alberi.

Una proprietà importante dei B-alberi è il fatto che si possa scegliere d in modo tale che i riferimenti ai d figli e le $d - 1$ chiavi memorizzate in un nodo possano trovar posto in un singolo blocco del disco, avendo come conseguenza che d sia proporzionale a B . Questa scelta ci consente di ipotizzare, nell'analisi delle operazioni di ricerca e modifica operanti su alberi (a, b) , che anche a e b siano proporzionali a B . Quindi, le funzioni $f(b)$ e $g(b)$ diventano $O(1)$, perché ci basta un solo trasferimento da disco per ogni accesso a un nodo effettuato durante una ricerca o una modifica.

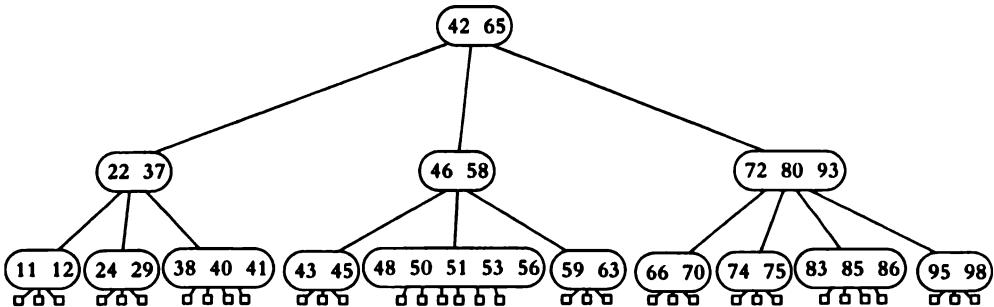


Figura 15.7: Un B -albero di ordine 6.

Come già osservato in precedenza, ogni ricerca o modifica richiede di esaminare al massimo $O(1)$ nodi per ogni livello dell'albero. Quindi, tutte le operazioni della mappa (ricerca, inserimento o rimozione) realizzata con un B -albero richiedono soltanto un numero di trasferimenti da disco $O(\log_{\lceil d/2 \rceil} n)$, cioè $O(\log n / \log B)$. Ad esempio, un'operazione di inserimento scende nel B -albero per individuare il nodo in cui inserire la nuova voce. Se il nodo si pone in una situazione di *overflow* (perché viene ad avere $d + 1$ figli) per effetto di questo inserimento, allora lo si sottopone a *divisione (split)*, generando due nodi aventi, rispettivamente, $\lfloor (d + 1)/2 \rfloor$ e $\lceil (d + 1)/2 \rceil$ figli. Questa procedura viene, poi, ripetuta al livello immediatamente superiore e continuerà al massimo per $O(\log_B n)$ livelli.

Analogamente, se un'operazione di rimozione genera una situazione di *underflow* in un nodo (che risulta avere $\lceil d/2 \rceil - 1$ figli), allora effettuiamo un *trasferimento* da uno dei nodi fratelli avente almeno $\lceil d/2 \rceil + 1$ figli, oppure una *fusione* di tale nodo con un suo fratello, ripetendo poi tale elaborazione nel genitore. Come con l'operazione di inserimento, questo potrà portare modifiche lungo i nodi che si visitano risalendo il B -albero, per un massimo di $O(\log_B n)$ livelli. Il fatto di aver imposto che ogni nodo interno abbia almeno $\lceil d/2 \rceil$ figli implica che ogni blocco del disco che viene usato per memorizzare un B -albero sia pieno almeno per metà. Quindi, possiamo affermare quanto segue.

Proposizione 15.2: *Un B -albero con n voci ha una complessità di I/O $O(\log_B n)$ per le operazioni di ricerca e modifica, e usa un numero di blocchi $O(n/B)$, essendo B la dimensione di un blocco.*

15.4 Ordinamento nella memoria esterna

Oltre alle strutture dati, come le mappe, che hanno bisogno di essere implementate nella memoria esterna, ci sono anche molti algoritmi che hanno la necessità di elaborare insiemi di dati di dimensioni troppo grandi per poter trovare posto completamente nella memoria interna. In questo caso, l'obiettivo è quello di risolvere il problema algoritmico usando il minor numero possibile di trasferimenti di blocchi. L'ambito più tradizionale per questi algoritmi che devono operare nella memoria esterna è il problema dell'ordinamento.

Merge-sort a più vie

Un modo efficiente per ordinare un insieme S di n oggetti archiviati nella memoria esterna prevede una semplice variante del famoso algoritmo merge-sort. L'idea principale su cui si basa questa variante è quella di fondere contemporaneamente molte liste già ordinate ricorsivamente, riducendo così il numero di livelli della ricorsione. In particolare, una descrizione ad alto livello di questo metodo di *merge-sort a più vie* (*multiway merge-sort*) prevede di suddividere S in d sottoinsiemi, S_1, S_2, \dots, S_d , aventi circa la stessa dimensione, per poi ordinare ricorsivamente ciascun sottoinsieme S_i e, poi, fondere simultaneamente tutte le d liste ordinate per generare una configurazione ordinata di S . Se riusciamo a effettuare la procedura di fusione usando soltanto $O(n/B)$ trasferimenti da disco, allora, per valori di n abbastanza grandi, il numero totale di trasferimenti richiesti da questo algoritmo soddisfa la relazione di ricorrenza seguente:

$$t(n) = d \cdot t(n/d) + cn/B,$$

per una qualche costante $c \geq 1$. Possiamo interrompere la ricorsione quando $n \leq B$, perché a quel punto ci basta effettuare il trasferimento di un solo blocco, portando nella memoria interna tutti gli oggetti e, quindi, ordinandoli con un algoritmo efficiente per la memoria interna. Quindi, la condizione conclusiva per $t(n)$ è:

$$t(n) = 1 \quad \text{se } n/B \leq 1.$$

Questo implica che una soluzione in forma chiusa di $t(n)$ è $O((n/B) \log_d (n/B))$, cioè:

$$O((n/B) \log(n/B)/\log d).$$

Quindi, se possiamo scegliere d in modo che sia $\Theta(M/B)$, dove M è la dimensione della memoria interna, allora nel caso peggiore il numero di trasferimenti di blocchi dalla memoria esterna eseguiti da questo algoritmo merge-sort a più vie sarà abbastanza basso. Per motivi che saranno discussi nel prossimo paragrafo, sceglieremo:

$$d = (M/B) - 1.$$

L'unico aspetto di questo algoritmo che ci rimane da specificare, ora, è il modo in cui si possa eseguire la fusione a d vie usando solamente $O(n/B)$ trasferimenti di blocchi.

15.4.1 Fusione a più vie

Nell'algoritmo merge-sort standard (visto nel Paragrafo 12.1), la procedura di fusione combina due sequenze ordinate in un'unica sequenza ordinata, prendendo ripetutamente l'elemento più piccolo tra i due che si trovano nella posizione iniziale delle rispettive liste. In una fusione a d vie, cerchiamo ripetutamente l'elemento minimo tra gli elementi iniziali delle d sequenze e lo inseriamo come elemento successivo nella sequenza risultante, continuando fino all'esaurimento degli elementi.

Trattandosi di un algoritmo di ordinamento che opera nella memoria esterna, se la memoria interna ha dimensione M e ogni blocco ha dimensione B , possiamo memorizzare

contemporaneamente nella memoria interna fino a M/B blocchi, quindi scegliamo appositamente $d = (M/B) - 1$, in modo che sia sempre possibile avere nella memoria interna un blocco per ciascuna delle d sequenze da fondere, e un ulteriore blocco che viene utilizzato per la sequenza risultante, come si può vedere nella Figura 15.8.

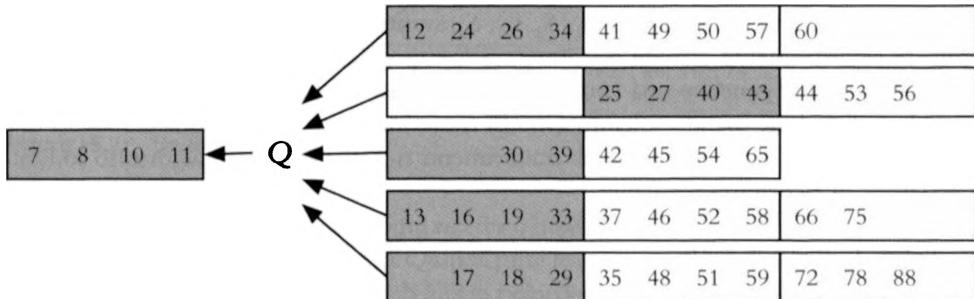


Figura 15.8: Un momento di una fusione a d vie, con $d = 5$ e $B = 4$. I blocchi che si trovano nella memoria interna sono quelli grigi.

In questo modo gli elementi minimi di ciascuna sequenza (che sono anche quelli iniziali) si trovano sempre nella memoria interna e, quando il blocco di una determinata sequenza viene svuotato, si richiede il successivo blocco di quella sequenza. Analogamente, usiamo un blocco della memoria interna per memorizzare la sequenza risultante, spostandolo nella memoria esterna e svuotandolo quando è pieno. In questo modo, il numero totale di trasferimenti eseguiti durante una singola fusione a d vie è $O(n/B)$, perché trasferiamo ciascun blocco della lista S_i una sola volta e scriviamo nella memoria esterna ciascun blocco della lista risultante S' una sola volta. In termini di tempo di elaborazione, la scelta del minimo tra d valori si può banalmente effettuare con $O(d)$ operazioni. Se volessimo dedicare a questo sotto-problema uno spazio $O(d)$ nella memoria interna, potremmo gestire una coda prioritaria che contenga sempre l'elemento minimo di ciascuna sequenza, eseguendo così ciascun passo della fusione in un tempo $O(\log d)$ mediante la rimozione dell'elemento minimo dalla coda prioritaria e l'inserimento nella stessa del successivo elemento prelevato dalla stessa sequenza. Quindi, il tempo di elaborazione relativo alle operazioni che coinvolgono la memoria interna per una fusione a d vie è $O(n \log d)$.

Proposizione 15.3: *Data una sequenza S basata su array e contenente n elementi, archiviata nella memoria esterna, possiamo ordinirla effettuando $O((n/B) \log(n/B)/\log(M/B))$ trasferimenti di blocchi e $O(n \log n)$ elaborazioni nella memoria interna, dove M è la dimensione della memoria interna e B è la dimensione di un blocco.*

15.5 Esercizi

Riepilogo e approfondimento

- R-15.1 Julia ha comprato un nuovo computer che usa numeri interi a 64 bit come indirizzi delle celle di memoria. Spiegare perché Julia non sarà mai in grado, nella sua vita, di aumentare la memoria principale del suo computer in modo che raggiunga la massima dimensione possibile, nell'ipotesi che per rappresentare bit distinti servano atomi distinti.
- R-15.2 Considerare una memoria cache inizialmente vuota costituita da quattro pagine. Elaborando la sequenza di richieste (2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3) con l'algoritmo di eliminazione LRU, quante volte si avrà una condizione di *page miss*, cioè di richiesta che *non* trova la pagina nella cache?
- R-15.3 Considerare una memoria cache inizialmente vuota costituita da quattro pagine. Elaborando la sequenza di richieste (2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3) con l'algoritmo di eliminazione FIFO, quante volte si avrà una condizione di *page miss*, cioè di richiesta che *non* trova la pagina nella cache?
- R-15.4 Considerare una memoria cache inizialmente vuota costituita da quattro pagine. Elaborando la sequenza di richieste (2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3) con l'algoritmo di eliminazione casuale, nel caso peggiore quante volte si avrà una condizione di *page miss*, cioè di richiesta che *non* trova la pagina nella cache? Elencare le scelte casuali fatte dall'algoritmo in tale caso peggiore.
- R-15.5 Descrivere in dettaglio gli algoritmi che effettuano l'inserimento e la rimozione di un elemento in un albero (a, b) .
- R-15.6 Nell'ipotesi che T sia un albero di ricerca a più vie in cui ogni nodo interno ha almeno cinque e al massimo otto figli, quali valori di a e b rendono T un albero (a, b) valido?
- R-15.7 Per quali valori di d l'albero T dell'esercizio precedente è un *B*-albero di ordine d ?
- R-15.8 Disegnare il risultato dell'inserimento, in un *B*-albero di ordine 7 inizialmente vuoto, delle voci aventi chiavi (4, 40, 23, 50, 11, 34, 62, 78, 66, 22, 90, 59, 25, 72, 64, 77, 39, 12), in questo ordine.

Creatività

- C-15.9 Descrivere un algoritmo efficiente che operi nella memoria esterna per eliminare tutte le voci duplicate in una lista con indice di dimensione n .
- C-15.10 Descrivere una struttura dati efficiente per implementare nella memoria esterna l'ADT "pila" in modo che il numero totale di trasferimenti da disco necessari a elaborare una sequenza di k operazioni push e pop sia $O(k/B)$.
- C-15.11 Descrivere una struttura dati efficiente per implementare nella memoria esterna l'ADT "coda" in modo che il numero totale di trasferimenti da disco necessari a elaborare una sequenza di k operazioni enqueue e dequeue sia $O(k/B)$.
- C-15.12 Descrivere una versione del tipo di dato astratto *PositionalList* (visto nel Paragrafo 7.3) che sia adatta alla memoria esterna, con dimensione dei blocchi uguale a B , in modo che una scansione di una lista di lunghezza n venga portata a termine nel caso peggiore effettuando $O(n/B)$ trasferimenti di blocchi, mentre tutti gli altri metodi dell'ADT richiedano un numero di trasferimenti $O(1)$.

- C-15.13 Modificare le regole che definiscono gli alberi rosso-nero in modo che ogni albero rosso-nero T abbia un corrispondente albero $(4, 8)$, e viceversa.
- C-15.14 Descrivere una versione modificata dell'algoritmo di inserimento in un B -albero in modo che, ogni volta che si crea un overflow a causa della divisione di un nodo w , le chiavi vengano distribuite tra tutti i fratelli di w , facendo in modo che ciascuno dei fratelli, alla fine, contenga circa lo stesso numero di chiavi (eventualmente innescando a cascata nel genitore di w l'esigenza di suddivisione). Qual è la frazione minima di ciascun blocco di memoria che sarà sempre piena usando questo schema?
- C-15.15 Un'altra possibile implementazione di mappa nella memoria esterna usa una *skip list*, allo scopo di raccogliere in singoli blocchi gruppi consecutivi di $O(B)$ nodi appartenenti a qualsiasi livello della skip list. In particolare, chiamiamo *B -skip list di ordine d* una tale rappresentazione di skip list, dove ogni blocco contiene almeno $\lceil d/2 \rceil$ nodi e al massimo d nodi. In questo caso sceglieremo d uguale al massimo numero di nodi appartenenti a uno stesso livello che possono trovar posto in un unico blocco. Descrivere come si debbano modificare gli algoritmi di inserimento e rimozione della skip list in modo che funzionino per questa nuova struttura e che diano luogo a un'altezza attesa $O(\log n / \log B)$.
- C-15.16 Descrivere come si possa usare un B -albero per implementare l'ADT "partizione", visto nel Paragrafo 14.7.3, in modo che le operazioni *union* e *find* usino, ciascuna, al massimo $O(\log n / \log B)$ trasferimenti da disco.
- C-15.17 Supponiamo che ci venga fornita una sequenza S di n elementi con numeri interi come chiavi, tale che alcuni elementi di S siano "di colore blu", mentre gli altri sono "di colore rosso". Inoltre, diciamo che un elemento rosso *e fa coppia* con un elemento blu f se hanno la stessa chiave. Descrivere un algoritmo efficiente, da utilizzare nella memoria esterna, per trovare tutte le coppie rosso-blu presenti in S . Quanti trasferimenti da disco sono necessari per l'esecuzione dell'algoritmo?
- C-15.18 Consideriamo il problema del *caching* delle pagine nel caso in cui la memoria cache possa contenere m pagine e venga fornita una sequenza P di n richieste relative a pagine selezionate in un insieme di $m + 1$ pagine possibili. Descrivere una strategia ottimale per l'algoritmo di eliminazione delle pagine e dimostrare che, in totale e partendo da una cache vuota, provoca al massimo $m + n/m$ eventi di *page miss*, cioè di pagina richiesta che non viene trovata nella cache.
- C-15.19 Descrivere un algoritmo efficiente che operi nella memoria esterna per determinare se, in un array contenente n numeri interi, esiste un valore che ricorre più di $n/2$ volte.
- C-15.20 Consideriamo la strategia di *caching* delle pagine basata sulla regola LFU (*least frequently used*, cioè "usata meno frequentemente"), dove, quando viene richiesta una nuova pagina che non sia presente nella cache piena, per inserirla si elimina la pagina che è stata richiesta meno frequentemente tra quelle presenti. In caso di parità, la strategia LFU elimina la pagina usata meno frequentemente che si trova nella cache da più tempo. Dimostrare che esiste una sequenza P di n richieste in risposta alla quale la strategia LFU non trova la pagina nella cache per un numero di volte $\Omega(n)$, usando una cache che può contenere m pagine, mentre l'algoritmo ottimale fallirà soltanto $O(m)$ volte.

- C-15.21 Supponiamo che, in un *B*-albero T di ordine d , invece di avere la funzione di ricerca del nodo $f(d) = 1$, si abbia $f(d) = \log d$. Come si modifica il tempo d'esecuzione asintotico per una ricerca in T ?

Progettazione

- P-15.22 Scrivere una classe Java che simuli gli algoritmi di gestione della memoria *best-fit*, *worst-fit*, *first-fit* e *next-fit*. Determinare sperimentalmente quale metodo si comporta meglio con diverse sequenze di richieste.
- P-15.23 Scrivere una classe Java che implementi tutti i metodi dell'ADT "mappa ordinata" mediante un albero (a, b) , dove a e b sono numeri interi costanti passati come parametri al costruttore.
- P-15.24 Implementare la struttura dati *B*-albero, ipotizzando blocchi di dimensione 1024 e numeri interi come chiavi. Verificare il numero di "trasferimenti da disco" necessari per elaborare una determinata sequenza di operazioni tipiche della mappa.

Note

Il lettore che fosse interessato allo studio dell'architettura dei sistemi gerarchici di memoria può fare riferimento al capitolo del libro di Burger *et al.* [20] oppure al libro di Hennessy e Patterson [44]. Il metodo di garbage collection *mark-sweep* che abbiamo descritto è soltanto uno dei molti algoritmi che risolvono il problema. Invitiamo il lettore interessato ad approfondire l'argomento consultando il libro di Jones e Lins [52]. Knuth [61] ha condotto un'ottima discussione sull'ordinamento e la ricerca nella memoria esterna. Il manuale di Gonnet e Baeza-Yates [38] confronta le prestazioni di alcuni algoritmi di ordinamento, molti dei quali sono dedicati alla memoria esterna. I *B*-alberi sono stati inventati da Bayer e McCreight [11] e Comer [24] presenta un'ottima visione panoramica dedicata a questa struttura dati. Anche i libri di Mehlhorn [71] e Samet [81] presentano una trattazione dei *B*-alberi e relative varianti. Aggarwal e Vitter [3] hanno studiato la complessità di I/O dell'ordinamento e i problemi connessi, determinandone limiti superiori e inferiori. Goodrich e coautori [40] hanno studiato la complessità di I/O di alcuni problemi di geometria computazionale. Il lettore interessato a ulteriori studi di algoritmi con I/O efficiente è invitato a leggere la panoramica scritta da Vittel [91].

A

Proprietà matematiche utili

In questa appendice esponiamo alcune proprietà matematiche utili, iniziando con alcune definizioni e proprietà del calcolo combinatorio.

Logaritmi e potenze

La funzione logaritmo è definita in questo modo:

$$\log_b a = c \quad \text{se} \quad a = b^c.$$

Per logaritmi e potenze, valgono le seguenti proprietà:

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_a a)/\log_a b$
5. $b^{\log_a c} = a^{\log_a b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a/b^c = b^{a-c}$

Inoltre, vale la seguente affermazione.

Proposizione A.1: Se $a > 0$, $b > 0$ e $c > a + b$, allora:

$$\log a + \log b < 2 \log c - 2.$$

Dimostrazione: È sufficiente dimostrare che $ab < c^2/4$. Possiamo scrivere:

$$\begin{aligned} ab &= \frac{a^2 + 2ab + b^2 - a^2 + 2ab - b^2}{4} = \\ &= \frac{(a+b)^2 - (a-b)^2}{4} \leq \frac{(a+b)^2}{4} < \frac{c^2}{4} \end{aligned}$$

■

La funzione *logaritmo naturale*, $\ln x = \log_e x$, con $e = 2.71828\dots$, è il valore della serie seguente:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Inoltre:

$$\begin{aligned} e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ \ln(1+x) &= x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \dots \end{aligned}$$

Molte sono le disuguaglianze utili che coinvolgono queste funzioni, in particolare citiamo le seguenti, che derivano dalle definizioni date.

Proposizione A.2: Se $x > -1$, allora:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x.$$

Proposizione A.3: Se $0 \leq x \leq 1$, allora:

$$1+x \leq e^x \leq \frac{1}{1-x}.$$

Proposizione A.4: Per qualunque coppia di numeri reali positivi x e n , si ha:

$$\left(1 + \frac{x}{n}\right)^n \leq e^x \leq \left(1 + \frac{x}{n}\right)^{n+x/2}$$

Funzioni e relazioni intere

Le funzioni *floor* e *ceiling* sono definite in questo modo:

1. Funzione *floor*: $\lfloor x \rfloor$ = il massimo numero intero non maggiore di x .
2. Funzione *ceiling*: $\lceil x \rceil$ = il minimo numero intero non minore di x .

Dati due numeri interi, $a \geq 0$ e $b > 0$, l'operazione *modulo* si indica con $a \bmod b$ ed è così definita:

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b.$$

Dato un numero intero $n > 0$, la funzione *fattoriale* è definita come:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1)n.$$

Il coefficiente binomiale, definito come:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

è uguale al numero di diverse *combinazioni* che si possono ottenere scegliendo k elementi diversi da una raccolta di n elementi (l'ordine non è importante). Il nome "coefficiente binomiale" deriva dalla sua presenza nella *espansione binomiale*:

$$(a+b)^n = \sum_{k=0}^n \binom{n}{k} a^k b^{n-k}.$$

Inoltre, sono valide le relazioni seguenti.

Proposizione A.5: Se $0 \leq k \leq n$, allora:

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \frac{n^k}{k!}.$$

Proposizione A.6: (*Approssimazione di Stirling*):

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \varepsilon(n)\right),$$

dove $\varepsilon(n)$ è $O(1/n^2)$.

La *successione di Fibonacci* è una successione numerica tale che $F_0 = 0$, $F_1 = 1$ e $F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$.

Proposizione A.7: Se F_n è definito dalla successione di Fibonacci, allora $F_n = \Theta(g^n)$, dove $g = (1 + \sqrt{5})/2$ è il cosiddetto rapporto aureo o sezione aurea.

Sommatorie

Ecco alcune sommatorie particolarmente utili.

Proposizione A.8: *Raccolta di un fattore comune:*

$$\sum_{i=1}^n af(i) = a \sum_{i=1}^n f(i),$$

a condizione che a non dipenda da i .

Proposizione A.9: *Inversione dell'ordine di esecuzione tra due sommatorie:*

$$\sum_{i=1}^n \sum_{j=1}^m f(i, j) = \sum_{j=1}^m \sum_{i=1}^n f(i, j).$$

Una sommatoria particolare è la *somma telescopica*:

$$\sum_{i=1}^n (f(i) - f(i-1)) = f(n) - f(0),$$

che compare spesso nell'analisi ammortizzata di una struttura dati o di un algoritmo.

Nel seguito riportiamo alcune proprietà utili che riguardano le sommatorie e che ricorrono frequentemente nell'analisi di strutture dati e algoritmi.

Proposizione A.10: $\sum_{i=1}^n i = n(n+1)/2$.

Proposizione A.11: $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$.

Proposizione A.12: Se $k \geq 1$ è un numero intero costante, allora:

$$\sum_{i=1}^n i^k \quad \text{è} \quad \Theta(n^{k+1}).$$

Un'altra sommatoria di uso frequente è la *sommatoria geometrica*, $\sum_{i=0}^n a^i$, definita per qualsiasi numero reale $0 < a \neq 1$.

Proposizione A.13:

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1},$$

per qualsiasi numero reale $0 < a \neq 1$.

Proposizione A.14:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

per qualsiasi numero reale $0 < a < 1$.

Esiste, poi, una combinazione di due delle sommatorie precedenti, chiamata *sommatoria linear-esponenziale*, che viene espressa in questo modo.

Proposizione A.15: Se $0 < a \neq 1$ e $n \geq 2$, allora:

$$\sum_{i=1}^n i a^i = \frac{a - (n+1)a^{(n+1)} + na^{(n+2)}}{(1-a)^2}.$$

L' n -esimo numero armonico, indicato con H_n , è così definito:

$$H_n = \sum_{i=1}^n \frac{1}{i}.$$

Proposizione A.16: Se H_n è l' n -esimo numero armonico, allora $H_n \in \ln n + \Theta(1)$.

Elementi di probabilità

Presentiamo qui alcune definizioni e proprietà elementari della teoria della probabilità. Iniziamo dicendo che qualunque affermazione che si occupi di probabilità è definita all'interno di uno *spazio dei campioni* S (o spesso, sinteticamente, "spazio campione") definito come l'insieme di tutti i possibili risultati di un esperimento, lasciando formalmente indefiniti, in questo contesto, i termini "esperimento" e "risultato di un esperimento".

Esempio A.17: Consideriamo un esperimento costituito da cinque lanci di una moneta. In questo caso lo spazio dei campioni contiene 2^5 diversi risultati, uno per ciascuna possibile disposizione dei singoli eventi che accadono ("testa" o "croce").

Lo spazio dei campioni può anche essere infinito, come vediamo nell'esempio che segue.

Esempio A.18: Consideriamo un esperimento costituito dal lancio di una moneta ripetuto fino a quando non esce "testa". Per questo esperimento lo spazio dei campioni è infinito, essendo ogni possibile risultato una sequenza contenente per k volte "croce", seguito da un unico "testa", con $k = 1, 2, 3, \dots$

Uno *spazio delle probabilità* è uno spazio campione S associato a una funzione di probabilità \Pr che mette in corrispondenza sottoinsiemi di S con numeri reali appartenenti all'intervallo $[0, 1]$. Si tratta di un'astrazione matematica che rappresenta il concetto di probabilità

che accadano determinati “eventi”. Dal punto di vista formale, ogni sottoinsieme A di S è chiamato **evento** e la funzione di probabilità \Pr deve avere le seguenti proprietà elementari, relativamente agli eventi definiti in S :

1. $\Pr(\emptyset) = 0$.
2. $\Pr(S) = 1$.
3. $0 \leq \Pr(A) \leq 1$, per ogni $A \subseteq S$.
4. Se $A \subseteq S$, $B \subseteq S$ e $A \cap B = \emptyset$, allora $\Pr(A \cup B) = \Pr(A) + \Pr(B)$.

Due eventi A e B si dicono *indipendenti* se:

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B).$$

Un insieme di eventi $\{A_1, A_2, \dots, A_n\}$ si dicono *mutuamente indipendenti* se:

$$\Pr(A_{i_1} \cap A_{i_2} \cap \dots \cap A_{i_k}) = \Pr(A_{i_1}) \Pr(A_{i_2}) \dots \Pr(A_{i_k}).$$

per qualsiasi sottoinsieme $\{A_{i_1}, A_{i_2}, \dots, A_{i_k}\}$.

La **probabilità** che si verifichi un evento A , *condizionata* dal fatto che si sia verificato l'evento B , è indicata con $\Pr(A|B)$ ed è definita come il rapporto:

$$\frac{\Pr(A \cap B)}{\Pr(B)},$$

nell'ipotesi che $\Pr(B) > 0$.

Nella teoria della probabilità, per gestire gli eventi si utilizzano le *variabili casuali*. Dal punto di vista intuitivo, le variabili casuali sono variabili i cui valori dipendono dal risultato di esperimenti. Dal punto di vista formale, una *variabile casuale* è una funzione X che mette in corrispondenza i risultati presenti in uno spazio campione S con numeri reali, mentre una *variabile casuale indicatore* è una variabile casuale che correla risultati a valori nell'insieme $\{0, 1\}$. Spesso, nell'analisi di algoritmi e strutture dati, usiamo una variabile casuale X per caratterizzare il tempo d'esecuzione di un algoritmo probabilistico. In tal caso, lo spazio campione S è definito da tutti i possibili valori generati dalle sorgenti casuali usate dall'algoritmo.

In merito a una variabile casuale, la proprietà più importante è il suo valore tipico, detto anche medio o, più spesso, “atteso” (*expected*). Il *valore atteso* di una variabile casuale X è definito come:

$$E(X) = \sum_x x \Pr(X = x),$$

dove la sommatoria è definita sull'intervallo dei valori assunti da X (che in questo caso si assume essere un insieme discreto).

Proposizione A.19 (Linearità del Valore Atteso): Se X e Y sono due variabili casuali e c è un numero, allora:

$$E(X + Y) = E(X) + E(Y) \quad \text{e} \quad E(cX) = cE(X).$$

Esempio A.20: Se X è una variabile casuale che mette in corrispondenza il risultato del lancio di due dadi equi (cioè un esperimento) con la somma del punteggio riportato nella faccia visibile dei due dadi, allora $E(X) = 7$.

Dimostrazione: Per dimostrare questa affermazione, definiamo X_1 e X_2 come variabili casuali corrispondenti al punteggio di ciascun dado, per cui $X_1 = X_2$ (cioè si tratta di due esemplari della stessa funzione) e $E(X) = E(X_1 + X_2) = E(X_1) + E(X_2)$. Ciascun risultato possibile del lancio di un dado equo accade con probabilità $1/6$, quindi:

$$E(X_i) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{7}{2},$$

per $i = 1, 2$. Di conseguenza, $E(X) = 7$. ■

Due variabili casuali X e Y sono *indipendenti* se:

$$\Pr(X = x | Y = y) = \Pr(X = x),$$

per qualsiasi coppia di numeri reali x e y .

Proposizione A.21: Se due variabili casuali X e Y sono indipendenti, allora:

$$E(XY) = E(X)E(Y).$$

Esempio A.22: Se X è una variabile casuale che mette in corrispondenza il risultato del lancio di due dadi equi (cioè un esperimento) con il prodotto del punteggio riportato nella faccia visibile dei due dadi, allora $E(X) = 49/4$.

Dimostrazione: Definiamo X_1 e X_2 come variabili casuali corrispondenti al punteggio di ciascun dado. Le variabili X_1 e X_2 sono chiaramente indipendenti, quindi:

$$E(X) = E(X_1 X_2) = E(X_1)E(X_2) = (7/2)^2 = 49/4.$$

Il limite superiore qui riportato e i corollari che ne conseguono sono noti come *limiti di Chernoff*.

Proposizione A.23: Sia X la somma di un numero finito di variabili casuali indipendenti che possono assumere i valori 0 o 1 e sia $\mu > 0$ il valore atteso di X . Per ogni valore $\delta > 0$, si ha:

$$\Pr(X > (1 + \delta)\mu) < \left[\frac{e^\delta}{(1 + \delta)^{(1 + \delta)}} \right]^\mu.$$

Alcune tecniche matematiche

Per confrontare i ritmi di crescita di funzioni diverse, a volte è utile applicare la regola seguente.

Proposizione A.24 (Regola di L'Hôpital): Se $\lim_{n \rightarrow \infty} f(n) = +\infty$ e $\lim_{n \rightarrow \infty} g(n) = +\infty$, allora $\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} f'(n)/g'(n)$, dove $f'(n)$ e $g'(n)$ indicano, rispettivamente, le derivate di $f(n)$ e $g(n)$.

Quando si cerca un limite superiore o inferiore per una sommatoria, spesso è utile *suddividere la sommatoria*, in questo modo:

$$\sum_{i=1}^n f(i) = \sum_{i=1}^j f(i) + \sum_{i=j+1}^n f(i).$$

Un'altra tecnica utile consiste nel *limitare una somma mediante un integrale*. Se f è una funzione non decrescente, allora, nell'ipotesi che i termini seguenti siano tutti definiti, si ha:

$$\int_{a-1}^b f(x) dx \leq \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x) dx.$$

Nell'analisi di algoritmi che usano la strategia *dividi-e-conquista*, si presenta solitamente una relazione di ricorrenza generale che ha questa forma:

$$T(n) = aT(n/b) + f(n),$$

dove $a \geq 1$ e $b > 1$ sono due valori costanti.

Proposizione A.25: Sia $T(n)$ la funzione definita in precedenza. Allora:

1. Se $f(n) \in O(n^{\log_b a - \epsilon})$, con $\epsilon > 0$ costante, allora $T(n) \in \Theta(n^{\log_b a})$.
2. Se $f(n) \in \Theta(n^{\log_b a} \log^k n)$, con $k \geq 0$ costante intera, allora $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$.
3. Se $f(n) \in \Omega(n^{\log_b a + \epsilon})$, con $\epsilon > 0$ costante, e se $af(n/b) \leq cf(n)$, allora $T(n) \in \Theta(f(n))$.

Questa proposizione è nota come *metodo principale (master method)* per la caratterizzazione asintotica delle relazioni di ricorrenza tipiche degli algoritmi di tipo *dividi-e-conquista*.

Bibliografia

- [1] H. Abelson, G.J. Sussman e J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press, 2nd ed., 1996.
- [2] G.M. Adel'son-Vel'skii e Y. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263-266, 1962. English translation in *Soviet Math. Doklady*, vol. 3, pp. 1259-1262.
- [3] A. Aggarwal e J.S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116-1127, 1988.
- [4] A.V. Aho, "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 255-300, Amsterdam: Elsevier, 1990.
- [5] A.V. Aho, J. E. Hopcroft e J.D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
- [6] A.V. Aho, J.E. Hopcroft e J.D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.
- [7] R.K. Ahuja, T.L. Magnanti e J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- [8] K. Arnold, J. Gosling e D. Holmes, *The Java Programming Language*. The Java Series, Upper Saddle River, NJ: Prentice Hall, 4th ed., 2006.
- [9] O. Barůvka, "O jistem problemu minimalním," *Práca Moravské Přírodovedecké Společnosti*, vol. 3, pp. 37-58, 1926. (in Czech).
- [10] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance," *Acta Informatica*, vol. 1, n. 4, pp. 290-306, 1972.
- [11] R. Bayer e McCreight, "Organization of large ordered indexes," *Acta Inform.*, vol. 1, pp. 173-189, 1972.
- [12] R.E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

Indice analitico

Simboli

!, !=, 24
%, 23
&&, 24
*, 23
*/ , 3
+, 17, 23
++, 24
--, 23, 24
/, 23
//, /*, 3
/***, 3, 48-50
<, <=, 24
< >, 87-88
=, 25
==, 24, 130
>, >=, 24
\, 22
||, 24
[], *Vedi array*
@, *Vedi javadoc*

A

abstract, 10, 76-77
abstract data type, *Vedi ADT*
AbstractBinaryTree, 307, 325
AbstractCollection, 301

AbstractHashMap, 406-408
AbstractList, 301
AbstractMap, 391-393
AbstractPriorityQueue, 350-351
AbstractQueue, 301
AbstractSortedMap, 414
AbstractTree, 301-304, 325
activation frame o record, 184, 677
adaptable priority queue, 375-380, 640
AdaptablePQEntry, 378-379
AdaptablePriorityQueue, 378
adapter pattern (adattatore), 222
Adel'son-Vel'skii, 461
adjacency list, 600, 603-604
adjacency map, 600, 605
adjacency matrix, 600, 606
AdjacencyMapGraph, 607-610
ADT (abstract data type), 59, 72
albero (generico), 299-300
albero binario, 306, 311
coda (queue), 228
coda doppia (deque), 237, 240
coda prioritaria (priority queue), 346-347
coda prioritaria flessibile, 375-380
grafo (graph), 599-600
insieme (set), 429-430
insieme ordinato (sorted set), 430

- lista con indice, 247-249
- lista posizionale, 262-263
- mappa (map), 388-389
- mappa ordinata (sorted map), 412-413
- multi-insieme (multiset), 431-432
- multi-mappa (multimap), 433-434
- partizione (partition), 658
- pila (stack), 216
- albero, 295-304, *Vedi anche* nodo,
 - attraversamento
 - ADT, 299-300
 - come grafo, 597
 - definizione, 296
 - implementazione, 318-319
 - livello, 308
 - ordinato, 298
 - sottoalbero, 297
 - stringa rappresentativa, 331
- albero (2, 4), 481-491
- albero (a, b), 689-691
- albero AVL, 461-469
- albero B, 691-692
- albero BFS, 621
- albero binario, 304-318, *Vedi anche* nodo,
 - attraversamento, albero di ricerca binario
 - ADT, 306, 311
 - completo, 355-356, 362-363
 - con array, 316-318
 - con struttura concatenata, 310-316
 - definizione, 304-305
 - diametro, 341
 - disegno di, 332-333
 - implementazione, 310-318
 - improprio, 304
 - numerazione per livelli, 317
 - proprio, 304
 - sottoalbero, 304
- albero DFS, 612-613
- albero di decisione, 304-305, 537
- albero di espressione, 305-306
- albero di ricerca, 443
 - a più vie, 481-484
- albero di ricerca bilanciato, 449, 454-461
 - ristrutturazione (di una terna di nodi), 455-461
 - rotazione, 455-461
- albero di ricerca binario, 323-324, 443-454
- albero merge-sort, 514-515, 520
- albero quick-sort, 526-529
- albero ricoprente, 597
- minimo (MST), 644-658
- albero rosso-nero, 491-504
- albero splay, 469-481
- albero trie, *Vedi* trie
- alfabeto, 16, 556-557
- algoritmo, 141
- altezza di un albero, 302-304
- ammortamento, *Vedi* analisi temporale
 - ammortizzata
- analisi temporale, *Vedi anche* tempo d'esecuzione
 - ammortizzata, 254-258
 - asintotica, 155-169
 - di ricorsioni, 193-196
- and, 24, 25
- Apache Commons, 432
- API (application programming interface), 72
- argomento sulla riga di comando, 15
- array, 97-115
 - associativo, *Vedi* mappa
 - bidimensionale, 111-115, 131-132
 - circolare, 231
 - clonazione di, 134-136
 - dinamico, 252-259
 - equivalenza tra, 131-132
 - generico, 89
 - in Java, 19-21
 - sparso, 291
- array list, *Vedi* lista con indice
- ArrayBlockingQueue, 276
- ArrayDeque, 239, 276
- ArrayIndexOutOfBoundsException, 20
- ArrayIterator, 273-274
- ArrayList, 249-251
 - in `java.util`, 103, 252, 271, 276, 278
 - iteratore per, 273-274
- ArrayQueue, 232
- Arrays, 105-106, 131-132, 280
- ArrayStack, 219-220
- ASCII, 16, 557, 576
- assegnazione, 25-26
- astrazione, 59
- attraversamento di un albero, 319-335
 - implementazione, 324-325
 - in ampiezza, 321-322, 327
 - in ordine simmetrico, 322-323, 327-328
 - in post-ordine, 320-321, 326-327
 - in pre-ordine, 320-321, 326
 - percorso di Eulero, 333-335
- attraversamento di un grafo, 610-623
 - in ampiezza, 620-623

in profondità, 611-620
 auto-boxing, 18-19
 auto-unboxing, 18-19
 AVL tree, *Vedi* albero AVL
 AVLTreeMap, 468-469

B

B-albero, 691-692
 Bag, 432
 BalanceableBinaryTree, 450, 459-461
 best-fit, 679-680
 binary search, 187-188, 413-414
 binary search tree, 323-324, 443-454
 binary tree, *Vedi* albero binario
 binarySearch, 106
 BinaryTree, 306-307
 binomiale (coefficiente), 701
 blocco (di codice), 1-2
 blocking (nelle strutture dati), 276-277
 blocking (nella memoria), 684-685
 boolean, 3, 18, 24
 Boolean, 18
 bottom-up, *Vedi* heap (costruzione bottom-up), ordinamento per fusione
 Boyer-Moore (algoritmo di), 559-563
 BFS (breadth-first search), 620-623
 break, 31, 36
 breakpoint, 53
 brute force, 544, 556, 558-559
 BSTNode, 459-460
 bubble sort, 291
 bucket, 395-396, 407, *Vedi anche* collisione, tabella hash
 bucket sort, 538-539, 543
 buffer, 19, 37
 byte, 3, 18
 Byte, 18
 byte code, 676

C

cache (memoria), 682-683
 caching, 684-685
 campo, *Vedi* variabile di esemplare
 carattere, 16, 22
 case, 31
 caso medio, 146-147
 caso peggiore, 146-147
 cast, 27-28, 83-86
 catch, 78-80
 cattura (di eccezione), 77-80

ceiling (parte intera superiore), 148, 155, 700
 ChainHashMap, 409-410
 char, 3, 18
 Character, 18
 character-jump (euristica), 559
 checked, 82-83
 Chernoff, limite di, 551
 chiave, *Vedi* coda prioritaria, mappa
 chiusura transitiva, 623-627
 chunk, 679
 ciclo (in un grafo), 596
 ciclo (nella programmazione), 31-35
 cifrario di Cesare, 108-111
 CircularlyLinkedList, 124
 CircularQueue, 235
 ClassCastException, 84
 classe, 1-2, 4-5, 57, 59
 annidata, 91-92
 astratta, 10, 75-77, 301
 che implementa un'interfaccia, 59, 72
 concreta, 72
 definizione, 8
 di base, 62
 esterna, 91
 final, 10
 interna, 91-92
 involturlo (*wrapper*), 18
 CLASSPATH, 48
 clonazione (di strutture dati), 133-137
 clone, 134
 Cloneable, 75, 134
 CloneNotSupportedException, 134
 clustering, 404
 coda (queue), 227-234
 ADT, 228
 circolare, 235-236
 con array, 230-233
 con lista concatenata, 234
 coda doppia (deque), 236-240
 ADT, 237, 240
 con array circolare, 238-239
 con lista concatenata, 239
 in java.util, 239-240
 coda prioritaria (priority queue), 345-354
 ADT, 346-347
 con heap, 356-366
 con lista non ordinata, 351-353
 con lista ordinata, 353-354
 flessibile, 375-380, 640
 implementazione, 347-354

per ordinare, 371-372
 priorità, 345-346
 codifica di hash, 396-400
 hashCode, 399-400
 per scorrimento ciclico, 398-399
 polinomiale, 397-398
 collaudo, 16, 51-52
Collection, 276, 279-280
Collections, 279-280
 collisione, 396
 clustering, 404
 concatenazione separata, 402
 doppio hashing, 404
 esplorazione (probing), 403-404, 410-412
 gestione di, 401-404
 indirizzamento aperto, 402-404
 commento, 2-3, 48-51
Comparable, 75, 349, 430
Comparator, 349, 430
 comparatore, 349-350
 complemento (bit per bit), 25
 complessità di I/O, 688
 composition pattern (composizione), 86, 282, 347
 compressione, *Vedi* funzione di compressione
 compressione del testo, 576-579
 concatenazione (tra stringhe), 17, 23, 28
 concatenazione separata, 402, 408-410
 concorrenza, 277
ConcurrentLinkedDeque, 276
ConcurrentLinkedQueue, 276
ConcurrentModificationException, 278
ConcurrentSkipListMap, 420
ConcurrentSkipListSet, 431
continue, 36.
 controllo dell'accesso, 8-9
 controllo di flusso, 29-39
 convenzioni stilistiche in Java, 50-51
 conversione di tipo (cast), 27-28, 83-86
 copia (profonda e superficiale), 133-136
 coppia chiave-valore, *Vedi* voce
copyOf/copyOfRange, 105
 cortocircuito, 24-25
 costante (funzione), 147-148
 costruttore, 6, 13-14
 della superclasse, 64
CRC (schede), 45-46
 crittografia, 108-111
 cubica (funzione), 151-152
currentTimeMillis, 107, 142-143

D

DAG, 627
 dati di attivazione, 184, 677
 de Morgan (legge di), 170
 debugging, 51-53
default, 31
deque, *Vedi* coda doppia
Deque, 237-238
 in `java.util`, 239-240, 276
 design pattern, 47, 60-61
 adapter, 222
 composition, 86, 282, 347
 factory method, 312, 459
 template method, 76, 430, 458
DFS (depth-first search), 611-620
 diagramma di flusso, 30
 diagramma di ricorsione, 183-184, 677
 diamante, 88
 digrafo, 594
Dijkstra (algoritmo di), 633-643
 dimostrazione, 169-173
 con un esempio/controesempio, 169
 con invarianti di ciclo, 172-173
 per contraddizione o assurdo, 170
 per contrapposizione, 169-170
 per induzione, 170-172
 divide and conquer (dividi e conquista), 513-514, 525
do-while, 32-33
 doppio hashing, 404
 doppio nero, 498-501
 doppio rosso, 494-497
 double, 3, 18
Double, 18
 double-ended queue, *Vedi* coda doppia
DoublyLinkedList, 128-130
 dynamic dispatch, 65
 dynamic programming, *Vedi* programmazione dinamica

E

eccezione, 77-83
Eclipse, 16, 47
 edge, *Vedi* lato (di un grafo)
Edge, 599
 edge list, 600-603
ElementIterator, 275-276, 325
else, 29-31
entry, *Vedi* voce
Entry, 347-348

in `java.util`, 389
`enum`, 21
`enumerazione`, 21, 31
`equals`, 24, 105, 130-132
`equazione di ricorrenza`, 521
`equivalezza (tra strutture dati)`, 130-133
`erasure`, 133
`ereditarietà`, 61-71, 74-76, 301
 multipla, 63, 74-75
`Error`, 81-82
`esemplare`, 4-6
`esplorazione (probing)`, 403-404, 410-412
`esponenziale (funzione)`, 153-154
`espressione`, 22-28
 albero di, 305-306
`estensione`, *Vedi ereditarietà, splaying*
`euristica`, 284, 559, 661
`Exception`, 81-82
`extends`, 63, 90

F

`factory method pattern`, 312, 459
`false`, 22
`fail fast (iteratore)`, 273, 277-278
`failure function`, 564-566
`fattore di bilanciamento`, 339
`fattore di carico o di occupazione`, 402, 404-405, 407, 409
`fattoriale (funzione)`, 182-184, 701
`Fibonacci (successione di)`, 69-70, 206-208, 701
`FIFO`, 228, 229
`File`, 191
`file system`, 189-192
`final`, 10
`finally`, 79
`firma (di metodo)`, 7, 11, 81
`first-fit`, 679-680
`float`, 3, 18
`Float`, 18
`floor (parte intera inferiore)`, 155, 700
`flowchart`, 30
`Floyd-Warshall (algoritmo di)`, 624-627
`fondamentale (tipo)`, 3-4, 18
`for`, 33-35, 271-272
`foresta`, 597
`forma chiusa`, 521
`forza bruta`, 544, 556, 558-559
`frame (activation frame)`, 184, 677
`frammentazione`, 679

`free list`, 679-680
`funzione di compressione`, 396, 400-401
`funzione di hash`, 395-396, *Vedi anche codifica di hash, funzione di compressione*
`funzione fallimento`, 564-566
`fusione`,
 a più vie, 693-694
 di array ordinati, 517-519
 in un albero (2, 4), 489-490

G

`garbage collection`, 221, 680-682
`Gauss (sommatoria di)`, 150-151
`gerarchia di ereditarietà`, 61, 66
 delle eccezioni, 81-83
`gerarchia di memoria`, 682-686
 gestione della memoria, 675-688
 cache, 683-686
 dinamica, 678
`Google Core Libraries for Java`, *Vedi Guava*
`grafo (graph)`, 593-600
 aciclico (DAG), 596
 ADT, 599-600
 completo, 664
 componenti connessi di, 597
 connesso, 597
 denso, 606
 fortemente connesso, 597
 implementazioni, 600-610
 misto, 594
 non orientato, 594
 orientato, 594
 orientato aciclico (DAG), 627
 pesato, 631-633
 raggiungibilità, 596, 623
 sparso, 606
`Graph`, 599-560
`greedy (metodo)`, 556, 578-579, 633
`Guava`, 432, 433

H

`hash table`, *Vedi tabella hash*
`hashCode`, 399-400
`HashMap`, 433-434
`HashSet`, 431
`heap`, 354-356, *Vedi anche memoria heap*
 altezza, 356
 costruzione bottom-up, 366-370
 definizione, 355-356
`down-heap bubbling`, 362

- implementazione, 363-365
- up-heap bubbling, 357-359
- heap sort**, 374-375, 542
- HeapAdaptablePriorityQueue**, 379-380
- HeapPriorityQueue**, 364-366
- Horner** (metodo di), 178
- HTML**, 48, 225-227, 242
- Huffman** (codifica di), 577-578

- I**
- IDE (integrated development environment)**, 16, 47-48, 53
- identificatore**, 2
- if**, 29-31
- IllegalArgumentException**, 80, 300
- import**, 43-44
- importazione** (di pacchetto), 42-44
- incapsulamento**, 8, 59
- IndexOutOfBoundsException**, 248
- indice**, 16, 19, 249
 - inverso, 575
- indicizzazione**, 575
- indirizzamento aperto**, 402-404
- indirizzo in memoria**, 5, 675
- induzione**, 170-172
- inferenza**, 88
- inizializzazione di variabile**, 4
- InnerEdge**, 607
- InnerVertex**, 607
- insertion sort**, 103-105, 280-281, 373-374, 541-542
- insieme (set)**, 429, 431
 - ADT, 429-430
- insieme dei massimi**, 418-419
- insieme ordinato (sorted set)**, 430-431
- instanceof**, 65, 84
- int**, 3, 18
- Integer**, 18
- interfaccia**, 59, 72-75
 - API, 72
 - cast, 85
- invariante di ciclo**, 172-173
- inversione (in una sequenza)**, 374
- involtuccio (classe)**, 18
- isomorfismo tra alberi**, 338
- istanza**, 4-6
- Iterable**, 271-272, 300
- Iterator**, 270, 273-274
- iteratore**, 270-276, 277-278
 - fail fast, 273, 277-278
- lazy ("pigro")**, 272
- snapshot**, 272

- J**
- java**, 47
- Java Collections Framework (JCF)**, 239, 276-280, 301
- Java operand stack**, 678
- Java runtime environment**, 6-7, 15, 676
- Java runtime stack**, 77, 676-677
- java.io**, 191
- java.lang**, 10, 63
- java.util**, 105, 276, 389
- java.util.concurrent**, 420
- javac**, 48
- javadoc**, 3, 48-50
- JCF**, Vedi Java Collections Framework
- jdb**, 53
- Josephus** (problema di), 235-236
- JUnit**, 16, 52
- JVM (Java Virtual Machine)**, 17, 77, 81, 676

- K**
- KeyIterable**, 392
- KeyIterator**, 392
- KMP (Knuth-Morris-Pratt, algoritmo di)**, 563-567
- Kruskal** (algoritmo di), 648-658

- L**
- lancio (di eccezione)**, 80-81
- Landis**, 461
- lato (di un grafo, edge)**, 593
 - auto-anello, 596
 - destinazione, 595
 - entrante, 595
 - multipli, 596
 - non orientato, 593
 - origine, 595
 - orientato, 593
 - paralleli, 596
 - peso, 631
 - uscente, 595
 - vertici terminali, 595
- lazy (iteratore)**, 272
- LCA (lowest common ancestor)**, 340-341
- legge di de Morgan**, 170
- letterale**, 22
- LIFO**, 215, 218
- lineare (funzione)**, 149-150

- link hopping**, 115
linked list, *Vedi* lista concatenata
LinkedBinaryTree, 311-316
LinkedBlockingDeque, 276
LinkedBlockingQueue, 276
LinkedCircularQueue, 235
LinkedList, 239, 276, 277, 278
LinkedPositionalList, 266-268
 iteratore per, 274-276
LinkedQueue, 224, 235
LinkedStack, 223
Linux/Unix, 48
Liskov (principio di), 65
List, 247-249
 in `java.util`, 247, 276, 278
lista, 247-249
lista con indice (array list), 249-252, 278
 ADT, 247-249
lista concatenata, 115
 circolare, 121-124
 clonazione, 136-137
 doppiamente, 125-130
 equivalenza tra, 132-133
 link hopping, 115
 semplicemente, 115-120
lista di adiacenze, 600, 603-604
lista di lati, 600-603
lista doppiamente concatenata, 125-130, *Vedi anche* lista concatenata
lista posizionale, 259-261, 278
 ADT, 262-264
 con array, 269-270
 con lista concatenata, 265-269
 posizione, 261, 264
lista semplicemente concatenata, 115-120, *Vedi anche* lista concatenata
 eliminazione dall'inizio, 118
 eliminazione dalla fine, 119
 implementazione, 119-120
 inserimento all'inizio, 116-117
 inserimento alla fine, 117-118
ListIterator, 277-278
livello (in un albero), 308
load factor, 402, 404-405, 407, 409
località,
 dei riferimenti, 284
 spaziale, 684-686
 temporale, 684-686
location-aware (voce/entry), 377-378
Locator, 662
log-asterisco (funzione), 662
log-lineare (funzione), 150
log-star (funzione), 662
logaritmo (funzione), 148-149, 699-700
long, 3, 18
Long, 18
look-up table, 395
looking-glass (euristica), 559
- M**
- macchina virtuale di Java (JVM)**, 17, 77, 81, 676
main, 1-2, 15, 78
map, *Vedi* mappa
Map, 390, 391-393
 in `java.util`, 389
MapEntry, 392
mappa (map), 387-390, *Vedi anche* mappa ordinata
 ADT, 388-389
 con `ArrayList`, 393-394
mappa di adiacenze, 600, 605
mappa ordinata (sorted map), 412-419, 443
 ADT, 412-413
 applicazioni, 417-419
 implementazione, 414-417
 skip list, 420-428
mark-sweep (algoritmo), 681-682
master method, 706
matrice (array bidimensionale), 111-115, 131-132
matrice di adiacenze, 600, 606
memoria, 675
 cache, 682-683
 di rete, 682-683
 esterna, 682-683
 heap, 678-679
 interna o principale, 682-683
 registri della CPU, 682-683
 virtuale, 685
merge sort, 513-524, 542
metodo, 1-2, 57
 astratto, 10, 75
 concreto, 75
 corpo, 11
 final, 10
 firma, 7, 11, 81
 generico, 90
 ridefinito o sovrascritto, 63-64
 static, 10
minimum spanning tree (MST), 644-658

- modificatore, 8-10
 modularità, 60
 modulo, 23, 701
 moltiplicazione matriciale a catena, 579-581
 motore di ricerca, 575
 move-to-front (euristica), 284-287
MST (minimum spanning tree), 644-658
 multi-insieme (multiset), 429, 431-432
 multi-mappa (multimap), 429, 432-434
Multimap, 433
Multiset, 432
- N**
- nanoTime**, 143
NavigableMap, 413
NavigableSet, 430
next-fit, 679-680
new, 6, 13, 20
Node, 265-266, 312-313,
 nodo (di un albero), 296-297, 310,
 altezza, 303-304
 antenato, 297
 bilanciato, 463
 discendente, 297
 figlio, 296, 304
 foglia, 297
 fratello, 297
 esterno, 297
 genitore, 296
 interno, 297
 profondità, 301-302
 radice, 296
 norma (di un vettore), 54
NoSuchElementException, 270
 not, 24
null, 6-7, 22, 221
NullPointerException, 6
Number, 84
NumberFormatException, 27, 80
 numerazione per livelli, 317, 355-356
 numeri,
 armonici, 703
 casuali, 106-107
 pseudocasuali, 106-108
- O**
- O-grande (notazione)**, 156-159
Object, 63, 86, 88
 equals, 130-131
 hashCode, 399-400
Observer, 75
- oggetto, 6-7, 57
 composito, 12
 radice, 680-682
 vivo, 680-682
Omega-grande (notazione), 159
open addressing, 402-404
OpenJDK, 406
 operatore, 23-26
 aritmetico, 23
 assegnazione, 25-26
 bit per bit, 25
 decremento, 24
 incremento, 24
 logico, 24-25
 precedenza, 26
 punto, 6-8, 9, 14
 or, 24, 25
 ordinamento, 103, 513-543
 a bolle, 291
 bucket sort, 538-539, 543
 con coda prioritaria, 371-372
 con heap, 374-375, 542
 confronto tra algoritmi, 541-543
 di una lista posizionale, 280-281
 ibrido, 536
 irregolare, 546
 lessicografico, 349
 limite inferiore asintotico, 536-538
 naturale, 349
 nella memoria esterna, 692-694
 per fusione, 513-524, 542
 per fusione a più vie, 693-694
 per inserimento, 103-105, 280-281,
 373-374, 541-542
 per selezione, 372-373
 quick sort, 525-536, 542
 quick sort probabilistico, 531-533, 542
 radix sort, 539-541, 543
 stabile, 539
 sul posto, 374-375, 533-534
 Tim sort, 542-543
 topologico, 628-631
 ottimalità statica, 481
 overflow,
 in albero (2, 4), 486
 in Java, 71
- P**
- pacchetto, 9, 42-44
 package, 42-43
 parametro (di un metodo), 11-13

- di tipo effettivo, 88
 - di tipo formale, 86
 - parola di memoria, 675
 - parola riservata, 2
 - parte intera (funzione), 148, 155, 700
 - Partition**, 662-663
 - partizione (partition), 658-663
 - ADT, 658
 - Patricia trie, 571-572
 - pattern matching, 556, 557
 - Boyer-Moore, 559-563
 - forza bruta, 558-559
 - KMP (Knuth-Morris-Pratt), 563-567
 - percorso (in un albero), 297
 - di Eulero, 333-335
 - percorso (in un grafo), 596
 - lunghezza, 632
 - orientato, 596
 - peso, 632
 - più breve, 631-643
 - semplice, 596
 - permutazione, 537
 - pianificazione circolare, 121-122, 235
 - pila (stack), 215-223
 - ADT, 216-218
 - con array, 219-221
 - con lista concatenata, 222-223
 - pila degli operandi, 678
 - pila di esecuzione, 77, 676-677
 - pivot, 525
 - scelta casuale, 531-533
 - mediana di tre valori, 535
 - polimorfismo, 64-65
 - polinomiale (funzione), 152-153
 - Position**, 264, 265, 300
 - PositionalList**, 264-265
 - PositionIterable**, 275-276
 - PositionIterator**, 274-276
 - posizione, 261, 264, 299
 - postfissa (notazione), 242
 - potenza (funzione), 199-201, 699-700
 - PQEntry, 350
 - precedenza tra operatori, 26
 - prefisso (di stringa), 557
 - prefix matching, 567
 - Prim-Jarník (algoritmo di), 645-648
 - principio di sostituzione, 65
 - PrintStream**, 37
 - priorità, 345-346
 - priority queue, *Vedi coda prioritaria*
 - PriorityQueue**, 348
 - in `java.util`, 370-371
 - private**, 9, 11
 - privato di pacchetto (accesso), 9
 - probabilità, 703-705
 - ProbeHashMap**, 410-412
 - probing, 403-404, 410-412
 - profondità di ricorsione, 208-209
 - progettazione del software, 44-53
 - orientata agli oggetti, 57-60
 - program counter, 677
 - programma, 1-2, 15-16
 - programmazione con tipi generici, 86-90
 - programmazione dinamica, 556, 579-585
 - allineamento di sequenze (LCS), 582-585
 - moltiplicazione matriciale a catena, 579-581
 - progressione, 66-67
 - aritmetica, 67-68
 - di Fibonacci, 69-70
 - geometrica, 68-69
 - protected**, 9
 - prune and search, 543-544
 - pseudocasuali (numeri), 106-108
 - pseudocodice, 46-47
 - public**, 8
 - puntatore, *Vedi riferimento*
- Q**
- quadratica (funzione), 150
 - queue, *Vedi coda*
 - Queue**, 228-229
 - in `java.util`, 229-230, 276
 - quick select deterministico, 551-552
 - quick select probabilistico, 544-546
 - quick sort, 525-536, 542
- R**
- radix sort, 539-541, 543
 - ramo (di un albero), 297
 - Random**, 106-108
 - ranking, 576
 - RBTreeMap**, 502-504
 - recoloring, 495, 498
 - red-black tree, 491-504
 - rehashing, 405
 - relazione,
 - d'ordine totale, 348
 - di equivalenza, 130-131
 - di ricorrenza, 521
 - gerarchica, 295

- r**
- `return`, 12, 35-36
 - ricerca,
 - binaria o per bisezione, 187-188, 413-414
 - lineare o sequenziale, 187
 - ricorsione, 181-182, 677-678
 - analisi, 193-196
 - binaria o doppia, 201-202
 - esempi, 182-192
 - in coda, 209-211
 - lineare, 197-201
 - multipla, 202-204
 - profondità di, 208-209
 - progettazione, 204-205
 - ridefinizione (di un metodo), 63-64
 - riduzione e ricerca, 543-544
 - riferimento, 5-6, 7-8
 - confronto tra, 24
 - rilassamento, 633-634
 - ristrutturazione (di una terna di nodi), 455-461, 464-466, 492, 498
 - rosso-nero (albero), 491-504
 - rotazione (in un albero di ricerca binario), *Vedi ristrutturazione*
 - round-robin, 121-122, 235
 - runtime stack, 77, 676-677
 - `RuntimeException`, 82-83
- S**
- `Scanner`, 38-39, 270
 - scelta del pivot, *Vedi pivot*
 - schema progettuale, *Vedi design pattern*
 - scorrimento (operatore), 25
 - search tree, *Vedi albero di ricerca*
 - selection sort, 372-373
 - selezione, 543-546
 - sentinella, 125-126, 338
 - separate chaining, 402, 408-410
 - set, *Vedi insieme*
 - `Set`, 429
 - `short`, 3, 18
 - `Short`, 18
 - shortest path, 631-643
 - `SinglyLinkedList`, 119-120
 - skip list, 420-428
 - smistamento dinamico, 65
 - snapshot, *Vedi iteratore*
 - sommatoria, 152, 154, 702-703
 - sorted map, *Vedi mappa ordinata*
 - sorted set, *Vedi insieme ordinato*
 - `SortedBag`, 432
 - `SortedMap`, 413
- `SortedMultiset`, 432
 - `SortedPriorityQueue`, 353-354
 - `SortedSet`, 430
 - `SortedTableMap`, 414-417
 - sorting, *Vedi ordinamento*
 - sotto-grafo, 597
 - sottoalbero, 297, 304
 - sottoclasse, 9, 62-63
 - sottestringa, 557
 - sovrascrittura (di un metodo), 63-64
 - spanning subgraph, 597
 - spanning tree, *Vedi albero ricoprente*
 - splay tree, 469-481
 - splaying, 469-474
 - `SplayTreeMap`, 474-476
 - split, 495
 - stack, *Vedi pila*
 - `Stack`, 217-218
 - in `java.util`, 218-219
 - static, 9-10
 - classe annidata, 91
 - statistica di ordine, 543
 - Stirling (approssimazione di), 701
 - `String`, 16-18, 557
 - stringa (di caratteri), 16-18, 556-557
 - nulla, 557
 - sottestringa, 557
 - `StringBuilder`, 17-18, 143-144, 259, 557
 - strong typing, 72
 - struttura dati, 141
 - stubbing, 52
 - suffisso (di stringa), 557
 - `super`, 64
 - superclasse, 62-63
 - `System.in`, 38
 - `System.out`, 37
 - switch, 31
- T**
- tabella di ricerca, 395
 - ordinata, 413
 - tabella hash, 394-412, *Vedi anche collisione, funzione di hash*
 - implementazione, 406-412
 - rehashing, 405
 - template method pattern, 76, 430, 458
 - tempo d'esecuzione, 141-142
 - analisi sperimentale, 142-145
 - atteso, 146
 - esponenziale, 161-162
 - nel caso medio, 146-147

nel caso peggiore, 146-147
 operazioni elementari, 145-146
testing, 16, 51-52
 Theta-grande (notazione), 159
this, 14-15, 64, 92
thread, 676
 multi-threaded, 676
 thread-safe, 276-277
throw, 80
trowable, 81-82
throws, 81
 tipizzazione forte, 72
tipo (di dato),
 astratto, *Vedi* ADT
 enumerativo, 21, 31
 generico, 87-89
 primitivo, 3-4, 18
 riferimento, *Vedi* riferimento
 vincolato, 90
token, 38-39
 Torri di Hanoi, 212
to**s**tring, 106
 transitive closure, 623-627
 trasferimento in un albero (2, 4), 489-490
 traversal, *Vedi* attraversamento
tree, *Vedi* albero
Tree, 300
Tree**M**ap, 449-453, 456-461
Tree**S**et, 431
trie, 567-576
 compresso, 571-572
 dei suffissi, 572-576
 standard, 568-571
true, 22
try, 78-80
type inference, 88

U

UML, 45-46, 62
unchecked, 83
 underflow in un albero (2, 4), 489-490
Unicode, 16, 22, 24, 557
Unix/Linux, 48
UnsupportedOperationException, 271
UnsortedPriorityQueue, 351-353
UnsortedTableMap, 393-394
URL (uniform resource locator), 387, 406

V

valore di hash, 396
 valore restituito, 12
ValueIterable, 393
ValueIterator, 392-393
 valutazione in cortocircuito, 24-25
variabile, 3
 assegnazione a, 26
 di esemplare, 11
 di tipo array, 20
 di tipo primitivo, 3
 di tipo riferimento, 5-6
 creditata, 63
 final, 10
 locale, 11
 oggetto, 6
 polimorfica, 65
 static, 9-10
Vertex, 599
 vertice (di un grafo, vertex), 593
 adiacenti, 595
 grado, 595
 raggiungibile, 596
vettore, *Vedi* lista con indice
visibilità, 8
voce (entry),
 in un multi-insieme, 432
 in una coda prioritaria, 346
 in una mappa, 387
 in una multi-mappa, 432
 location-aware, 377-378
void, 12

W

while, 32
Windows, 48
worst-fit, 679-680
wraparound, 233
wrapper (classe), 18

X, Z

XML, 225
zig, 471-474
zig-zag, 470-474
zig-zig, 470-474