Wprowadzenie do

JAVY

PROGRAMOWANIE I STRUKTURY DANYCH

WYDANIE XII

Y. DANIEL LIANG

Helion

Tytuł oryginału: Introduction to Java Programming and Data Structures, Comprehensive Version (12th Edition)

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-7082-1

Authorized translation from the English language edition, entitled INTRODUCTION TO JAVA PROGRAMMING AND DATA STRUCTURES, COMPREHENSIVE VERSION, 12th Edition by Y. LIANG, published by Pearson Education, Inc, publishing as Pearson, Copyright © 2020, 2018, 2015 by Pearson Education, Inc. or its affiliates, 221 River Street, Hoboken, NJ 07030.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
POLISH language edition published by Helion SA, Copyright © 2021.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

PEARSON, ALWAYS LEARNING, and MYLAB are exclusive trademarks owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem: *ftp://ftp.helion.pl/przyklady/wpjav12.zip*

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wpjav12
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa 19

Rozdział 1. Wprowadzenie do komputerów, programów i JavyTM 27

1.1. Wprowadzenie 28
1.2. Czym jest komputer? 28
1.3. Języki programowania 33
1.4. Systemy operacyjne 36
1.5. Java, sieć WWW i inne zagadnienia 37
1.6. Specyfikacja Javy, API, JDK, JRE i IDE 38
1.7. Prosty program w Javie 39
1.8. Tworzenie, kompilowanie i wykonywanie programu w Javie 42
1.9. Styl programowania i dokumentacja 46
1.10. Błędy w programach 47
1.11. Tworzenie programów za pomocą środowiska NetBeans 51
1.12. Tworzenie programów w Javie z użyciem środowiska Eclipse 54

Rozdział 2. Podstawy programowania 61

2.1. Wprowadzenie 62
2.2. Pisanie prostego programu 62
2.3. Wczytywanie danych wejściowych z konsoli 65
2.4. Identyfikatory 68
2.5. Zmienne 69
2.6. Instrukcje przypisania i wyrażenia przypisania 70
2.7. Stałe nazwane 72
2.8. Konwencje nazewnicze 73
2.9. Liczbowe typy danych i operacje na nich 73
2.10. Literały liczbowe 78
2.11. Narzędzie JShell 79
2.12. Obliczanie wyrażeń i priorytety operatorów 82
2.13. Studium przypadku: wyświetlanie aktualnego czasu 83
2.14. Złożone operatory przypisania 85
2.15. Operatory inkrementacji i dekrementacji 86
2.16. Konwersje typów liczbowych 88
2.17. Proces rozwoju oprogramowania 91
2.18. Studium przypadku: przeliczanie kwot pieniędzy na mniejsze nominały 95
2.19. Często występujące błędy i pułapki 97

Kup ksi k

Pole ksi k

Rozdział 3. Instrukcje sterujące **109**

3.1. Wprowadzenie 110
3.2. Typ danych boolean oraz wartości i wyrażenia logiczne 110
3.3. Instrukcje if 112
3.4. Instrukcje if-else z dwiema ścieżkami 115
3.5. Zagnieżdżone instrukcje if i instrukcje if-else z wieloma ścieżkami 116
3.6. Typowe błędy i pułapki 118
3.7. Generowanie liczb losowych 122
3.8. Studium przypadku: obliczanie wskaźnika BMI 124
3.9. Studium przypadku: obliczanie podatków 126
3.10. Operatory logiczne 129
3.11. Studium przypadku: wykrywanie roku przestępnego 133
3.12. Studium przypadku: loteria 134
3.13. Instrukcje switch 136
3.14. Operatory warunkowe 140
3.15. Priorytety i łączność operatorów 141
3.16. Debugowanie 142

Rozdział 4. Funkcje matematyczne, znaki i łańcuchy znaków **157**

4.1. Wprowadzenie 158
4.2. Standardowe funkcje matematyczne 158
4.3. Typ danych char i jego operacje 163
4.4. Typ String 169
4.5. Studia przypadku 178
4.6. Formatowanie danych wyjściowych w konsoli 185

Rozdział 5. Pętle **199**

5.1. Wprowadzenie 200
5.2. Pętla while 200
5.3. Studium przypadku: zgadywanie liczb 203
5.4. Strategie projektowania pętli 206
5.5. Sterowanie pętlą na podstawie potwierdzenia od użytkownika lub wartości wartownika 209
5.6. Pętla do-while 212
5.7. Pętla for 214
5.8. Której pętli użyć? 218
5.9. Pętle zagnieżdżone 220
5.10. Minimalizowanie błędów numerycznych 222
5.11. Studia przypadków 224
5.12. Słowa kluczowe break i continue 229
5.13. Studium przypadku: wykrywanie palindromów 232
5.14. Studium przypadku: wyświetlanie liczb pierwszych 234

Kup ksi k

Pole ksi k

Rozdział 6. Metody 249

6.1. Wprowadzenie 250
6.2. Definiowanie metody 251
6.3. Wywoływanie metody 252
6.4. Metody void i metody zwracające wartość 255
6.5. Przekazywanie argumentów przez wartość 258
6.6. Pisanie modułowego kodu 262
6.7. Studium przypadku: przekształcanie liczb szesnastkowych na dziesiętne 264
6.8. Przeciążanie metod 266
6.9. Zasięg zmiennych 269
6.10. Studium przypadku: generowanie losowych znaków 271
6.11. Abstrakcja w postaci metody i stopniowe uszczegóławianie kodu 273

Rozdział 7. Tablice jednowymiarowe 295

7.1. Wprowadzenie 296
7.2. Podstawowe informacje o tablicach 296
7.3. Studium przypadku: analizowanie liczb 303
7.4. Studium przypadku: talia kart 304
7.5. Kopiowanie tablic 306
7.6. Przekazywanie tablic do metod 308
7.7. Zwracanie tablicy przez metodę 311
7.8. Studium przypadku: zliczanie wystąpień każdej litery 312
7.9. Listy argumentów o zmiennej długości 315
7.10. Wyszukiwanie w tablicach 316
7.11. Sortowanie tablic 320
7.12. Klasa Arrays 322
7.13. Argumenty wiersza poleceń 324

Rozdział 8. Tablice wielowymiarowe 339

8.1. Wprowadzenie 340
8.2. Podstawy tablic dwuwymiarowych 340
8.3. Przetwarzanie tablic dwuwymiarowych 343
8.4. Przekazywanie tablic dwuwymiarowych do metod 345
8.5. Studium przypadku: ocena testu wielokrotnego wyboru 347
8.6. Studium przypadku: znajdowanie pary najbliższych punktów 349
8.7. Studium przypadku: sudoku 351
8.8. Tablice wielowymiarowe 354

Rozdział 9. Obiekty i klasy 375

9.1. Wprowadzenie 376
9.2. Definiowanie klas służących do tworzenia obiektów 376
9.3. Przykład: definiowanie klas i tworzenie obiektów 378

Kup ksi k Pole ksi k

9.4. Tworzenie obiektów z użyciem konstruktorów 383
9.5. Używanie obiektów za pomocą zmiennych referencyjnych 385
9.6. Używanie klas z biblioteki Javy 389
9.7. Zmienne, stałe i metody statyczne 392
9.8. Modyfikatory widoczności 398
9.9. Hermetyzacja pól 400
9.10. Przekazywanie obiektów do metod 402
9.11. Tablica obiektów 407
9.12. Niemodyfikowalne obiekty i klasy 409
9.13. Zasięg zmiennych 411
9.14. Referencja this 412

Rozdział 10. Myślenie obiektowe 421

10.1. Wprowadzenie 422
10.2. Abstrakcja w postaci klasy i hermetyzacja 422
10.3. Myślenie w sposób obiektowy 426
10.4. Relacje między klasami 429
10.5. Studium przypadku: projektowanie klasy Course 432
10.6. Studium przypadku: projektowanie klasy reprezentującej stos 434
10.7. Przetwarzanie wartości typów podstawowych jako obiektów 437
10.8. Automatyczna konwersja między typami podstawowymi a typami nakładkowymi 441
10.9. Klasy BigInteger i BigDecimal 442
10.10. Klasa String 443
10.11. Klasy StringBuilder i StringBuffer 451

Rozdział 11. Dziedziczenie i polimorfizm 467

11.1. Wprowadzenie 468
11.2. Nadklasy i podklasy 468
11.3. Używanie słowa kluczowego super 474
11.4. Przesłanianie metod 478
11.5. Przesłanianie a przeciążanie 478
11.6. Klasa Object i metoda toString() 480
11.7. Polimorfizm 481
11.8. Wiązanie dynamiczne 482
11.9. Rzutowanie obiektów i operator instanceof 486
11.10. Metoda equals z klasy Object 490
11.11. Klasa ArrayList 491
11.12. Przydatne metody dotyczące list 497
11.13. Studium przypadku: niestandardowa klasa reprezentująca stos 498
11.14. Dane i metody z modyfikatorem protected 500
11.15. Zapobieganie rozszerzaniu klas i przesłanianiu metod 502

Kup ksi k Pole ksi k

Rozdział 12. Obsługa wyjątków i tekstowe operacje wejścia – wyjścia 511

12.1. Wprowadzenie 512
12.2. Omówienie obsługi wyjątków 512
12.3. Typy wyjątków 518
12.4. Deklarowanie, zgłaszanie i przechwytywanie wyjątków 520
12.5. Klauzula finally 529
12.6. Kiedy stosować wyjątki? 531
12.7. Ponowne zgłaszanie wyjątków 532
12.8. Łańcuch wyjątków 532
12.9. Definiowanie niestandardowych klas wyjątków 533
12.10. Klasa File 537
12.11. Plikowe operacje wejścia – wyjścia 539
12.12. Wczytywanie danych z internetu 547
12.13. Studium przypadku: robot internetowy 549

Rozdział 13. Klasy abstrakcyjne i interfejsy 559

13.1. Wprowadzenie 560
13.2. Klasy abstrakcyjne 560
13.3. Studium przypadku: klasa abstrakcyjna Number 565
13.4. Studium przypadku: Calendar i GregorianCalendar 568
13.5. Interfejsy 570
13.6. Interfejs Comparable 575
13.7. Interfejs Cloneable 579
13.8. Interfejsy a klasy abstrakcyjne 584
13.9. Studium przypadku: klasa Rational 588
13.10. Wskazówki dotyczące projektowania klas 594

Rozdział 14. Podstawy platformy JavaFX 603

14.1. Wprowadzenie 604
14.2. JavaFX a Swing i AWT 604
14.3. Podstawowa struktura programu używającego JavaFX 604
14.4. Panele, grupy, kontrolki interfejsu użytkownika i kształty 607
14.5. Wiązanie właściwości 611
14.6. Wspólne właściwości i metody węzłów 614
14.7. Klasa Color 616
14.8. Klasa Font 617
14.9. Klasy Image i ImageView 619
14.10. Panele i grupy 622
14.11. Kształty 631
14.12. Studium przypadku: klasa ClockPane 645

Kup ksi k Pole ksi k

Rozdział 15. Programowanie sterowane zdarzeniami i animacje 659

15.1. Wprowadzenie 660
15.2. Zdarzenia i źródła zdarzeń 662
15.3. Rejestrowanie obiektów obsługi zdarzeń i obsługa zdarzeń 664
15.4. Klasy wewnętrzne 668
15.5. Anonimowe wewnętrzne klasy obsługi zdarzeń 669
15.6. Upraszczanie obsługi zdarzeń z użyciem wyrażeń lambda 672
15.7. Studium przypadku: kalkulator kredytowy 677
15.8. Zdarzenia związane z myszą 679
15.9. Zdarzenia związane z klawiszami 680
15.10. Odbiorniki dla obiektów obserwowalnych 684
15.11. Animacje 687
15.12. Studium przypadku: odbijająca się kulka 695
15.13. Studium przypadku: mapa Stanów Zjednoczonych 699

Rozdział 16. Kontrolki i multimedia JavaFX 713

16.1. Wprowadzenie 714
16.2. Labeled i Label 714
16.3. Button 717
16.4. CheckBox 719
16.5. RadioButton 722
16.6. TextField 725
16.7. TextArea 727
16.8. ComboBox 730
16.9. ListView 733
16.10. ScrollBar 737
16.11. Slider 740
16.12. Studium przypadku: tworzenie gry w kółko i krzyżyk 743
16.13. Wideo i dźwięk 748
16.14. Studium przypadku: flagi i hymny narodowe 752

Rozdział 17. Binarne operacje wejścia – wyjścia 765

17.1. Wprowadzenie 766
17.2. Jak tekstowe operacje wejścia – wyjścia są obsługiwane w Javie? 766
17.3. Tekstowe a binarne operacje I/O 767
17.4. Klasy binarnych operacji I/O 768
17.5. Studium przypadku: kopiowanie plików 779
17.6. Zapis i odczyt obiektów 781
17.7. Pliki o dostępie swobodnym 786

Kup ksi k

Pole ksi k

Rozdział 18. Rekurencja 795

18.1. Wprowadzenie 796
18.2. Studium przypadku: obliczanie silni 796
18.3. Studium przypadku: obliczanie liczb Fibonacciego 800
18.4. Rozwiązywanie problemów z użyciem rekurencji 803
18.5. Rekurencyjne metody pomocnicze 804
18.6. Studium przypadku: obliczanie wielkości katalogu 807
18.7. Studium przypadku: wieże Hanoi 809
18.8. Studium przypadku: fraktale 813
18.9. Rozwiązania rekurencyjne i iteracyjne 816
18.10. Rekurencja ogonowa 817

Rozdział 19. Typy generyczne 829

19.1. Wprowadzenie 830
19.2. Powody i zalety stosowania typów generycznych 830
19.3. Definiowanie klas i interfejsów generycznych 833
19.4. Metody generyczne 834
19.5. Studium przypadku: sortowanie tablicy obiektów 836
19.6. Typy surowe i zgodność wstecz 838
19.7. Typy generyczne z wyrażeniami wieloznacznymi 840
19.8. Wymazywanie typów i zastrzeżenia dotyczące typów generycznych 842
19.9. Studium przypadku: generyczna klasa reprezentująca macierze 845

Rozdział 20. Listy, stosy, kolejki i kolejki priorytetowe 853

20.1. Wprowadzenie 854
20.2. Kolekcje 854
20.3. Iteratory 858
20.4. Używanie metody forEach 860
20.5. Listy 861
20.6. Interfejs Comparator 866
20.7. Statyczne metody list i kolekcji 871
20.8. Studium przypadku: odbijające się kulki 874
20.9. Klasy Vector i Stack 878
20.10. Kolejki i kolejki priorytetowe 880
20.11. Studium przypadku: przetwarzanie wyrażeń 883

Rozdział 21. Zbiory i odwzorowania 895

21.1. Wprowadzenie 896
21.2. Zbiory 896
21.3. Porównywanie wydajności zbiorów i list 905
21.4. Studium przypadku: zliczanie słów kluczowych 907

Kup ksi k Pole ksi k

21.5. Odwzorowania 909
21.6. Studium przypadku: wystąpienia słów 914
21.7. Jednoelementowe i niemodyfikowalne kolekcje i odwzorowania 916

Rozdział 22. Pisanie wydajnych algorytmów 921

22.1. Wprowadzenie 922
22.2. Pomiar wydajności algorytmów za pomocą notacji dużego O 922
22.3. Przykłady: wyznaczanie dużego O 924
22.4. Analizowanie złożoności czasowej algorytmów 928
22.5. Wyznaczanie liczb Fibonacciego z wykorzystaniem programowania dynamicznego 931
22.6. Znajdowanie największych wspólnych dzielników za pomocą algorytmu Euklidesa 934
22.7. Wydajne algorytmy do znajdowania liczb pierwszych 938
22.8. Znajdowanie pary najbliższych punktów metodą dziel i rządź 945
22.9. Rozwiązywanie problemu ośmiu hetmanów za pomocą algorytmu z nawrotami 948
22.10. Geometria obliczeniowa: znajdowanie otoczki wypukłej 951
22.11. Dopasowywanie łańcuchów znaków 954

Rozdział 23. Sortowanie 973

23.1. Wprowadzenie 974
23.2. Sortowanie przez wstawianie 974
23.3. Sortowanie bąbelkowe 977
23.4. Sortowanie przez scalanie 979
23.5. Sortowanie szybkie 983
23.6. Sortowanie przez kopcowanie 987
23.7. Sortowanie kubełkowe i pozycyjne 996
23.8. Sortowanie zewnętrzne 997

Rozdział 24. Implementowanie list, stosów, kolejek i kolejek priorytetowych 1011

24.1. Wprowadzenie 1012
24.2. Standardowe operacje na listach 1012
24.3. Listy tablicowe 1016
24.4. Listy powiązane 1023
24.5. Stosy i kolejki 1038
24.6. Kolejki priorytetowe 1042

Rozdział 25. Binarne drzewa poszukiwań 1049

25.1. Wprowadzenie 1050
25.2. Podstawy binarnych drzew poszukiwań 1050
25.3. Reprezentowanie drzew BST 1051
25.4. Wyszukiwanie elementu 1052

Kup ksi k Pole ksi k

25.5. Wstawianie elementu do drzewa BST 1052
25.6. Przechodzenie drzewa 1054
25.7. Klasa BST 1055
25.8. Usuwanie elementów z drzewa BST 1065
25.9. Wizualizowanie drzew i architektura MVC 1071
25.10. Iteratory 1075
25.11. Studium przypadku: kompresja danych 1077

Rozdział 26. Drzewa AVL 1087

26.1. Wprowadzenie 1088
26.2. Wyważanie drzew 1088
26.3. Projektowanie klas dla drzew AVL 1091
26.4. Przesłanianie metody insert 1092
26.5. Implementowanie rotacji 1094
26.6. Implementowanie metody delete 1094
26.7. Klasa AVLTree 1095
26.8. Testowanie klasy AVLTree 1101
26.9. Analiza złożoności czasowej operacji w drzewach AVL 1104

Rozdział 27. Haszowanie 1109

27.1. Wprowadzenie 1110
27.2. Czym jest haszowanie? 1110
27.3. Funkcje haszujące i skróty 1111
27.4. Zarządzanie kolizjami z użyciem otwartego adresowania 1113
27.5. Zarządzanie kolizjami metodą łańcuchową 1117
27.6. Współczynnik wypełnienia i ponowne haszowanie 1118
27.7. Implementowanie odwzorowania z użyciem haszowania 1120
27.8. Implementowanie zbioru z użyciem haszowania 1130

Rozdział 28. Grafy i ich zastosowania 1141

28.1. Wprowadzenie 1142
28.2. Podstawowa terminologia z obszaru grafów 1143
28.3. Reprezentowanie grafów 1144
28.4. Modelowanie grafów 1151
28.5. Wizualizowanie grafów 1161
28.6. Przechodzenie grafu 1165
28.7. Przeszukiwanie w głąb 1166
28.8. Studium przypadku: problem połączonych kół 1170
28.9. Przeszukiwanie wszerz 1173
28.10. Studium przypadku: problem dziewięciu monet 1176

Rozdział 29. Grafy ważone i ich zastosowania 1189

29.1. Wprowadzenie 1190
29.2. Reprezentowanie grafów ważonych 1191
29.3. Klasa WeightedGraph 1193
29.4. Minimalne drzewa rozpinające 1201
29.5. Znajdowanie najkrótszych ścieżek 1209
29.6. Studium przypadku: problem dziewięciu monet z wagami 1217

Rozdział 30. Operacje agregujące dla strumieni do przetwarzania kolekcji 1229

30.1. Wprowadzenie 1230
30.2. Potoki 1230
30.3. IntStream, LongStream i DoubleStream 1237
30.4. Równoległe strumienie 1239
30.5. Redukcja strumienia z użyciem metody reduce 1242
30.6. Redukcja strumieni za pomocą metody collect 1245
30.7. Grupowanie elementów za pomocą kolektora groupingBy 1248
30.8. Studium przypadku 1251

Rozdziały 31 - 37 są dostępne online pod adresem
https://ftp.helion.pl/przyklady/wpja12.zip

Rozdział 31. Zaawansowane zagadnienia z obszaru JavaFX i FXML 1263

31.1. Wprowadzenie 1264
31.2. Style CSS z JavaFX 1264
31.3. Klasy QuadCurve, CubicCurve i Path 1266
31.4. Modyfikowanie współrzędnych 1271
31.5. Pędzle 1276
31.6. Menu 1280
31.7. Menu kontekstowe 1285
31.8. Panele SplitPane 1287
31.9. Panele TabPane 1290
31.10. TableView 1292
31.11. Pisanie programów dla architektury JavaFX za pomocą języka FXML 1299

Rozdział 32. Wielowątkowość i programowanie równoległe 1317

32.1. Wprowadzenie 1318
32.2. Zagadnienia związane z wątkami 1318
32.3. Tworzenie zadań i wątków 1319
32.4. Klasa Thread 1322
32.5. Animacja z użyciem wątków i metody Platform.runLater 1325

Kup ksi k Pole ksi k

32.6. Pule wątków 1327
32.7. Synchronizacja wątków 1329
32.8. Synchronizacja z użyciem blokad 1333
32.9. Współdziałanie między wątkami 1335
32.10. Studium przypadku: wzorzec producent/konsument 1340
32.11. Kolejki z blokowaniem 1343
32.12. Semafory 1346
32.13. Unikanie zakleszczenia 1347
32.14. Stany wątków 1348
32.15. Synchronizowane kolekcje 1349
32.16. Programowanie równoległe 1350

Rozdział 33. Sieci 1361

33.1. Wprowadzenie 1362
33.2. Model klient-serwer 1362
33.3. Klasa InetAddress 1370
33.4. Obsługa wielu klientów 1371
33.5. Wysyłanie i przyjmowanie obiektów 1374
33.6. Studium przypadku: kółko i krzyżyk w środowisku rozproszonym 1378

Rozdział 34. Umiędzynarodowienie 1397

34.1. Wprowadzenie 1398
34.2. Klasa Locale 1398
34.3. Wyświetlanie daty i czasu 1401
34.4. Formatowanie liczb 1413
34.5. Pakiety zasobów 1420
34.6. Kodowanie znaków 1427

Rozdział 35. Drzewa 2-3-4 i B-drzewa 1433

35.1. Wprowadzenie 1434
35.2. Projektowanie klas na potrzeby drzew 2-3-4 1435
35.3. Wyszukiwanie elementu 1435
35.4. Wstawianie elementu w drzewie 2-3-4 1437
35.5. Usuwanie elementów z drzewa 2-3-4 1439
35.6. Odwiedzanie elementów w drzewie 2-3-4 1443
35.7. Implementowanie klasy Tree24 1445
35.8. Testowanie klasy Tree24 1453
35.9. Analiza złożoności czasowej 1455
35.10. B-drzewo 1457

Kup ksi k

Pole ksi k

Rozdział 36. Drzewa czerwono-czarne 1463

36.1. Wprowadzenie 1464
36.2. Konwersja między drzewami czerwono-czarnymi a drzewami 2-3-4 1464
36.3. Projektowanie klas drzew czerwono-czarnych 1466
36.4. Przesłanianie metody insert 1467
36.5. Przesłanianie metody delete 1472
36.6. Implementowanie klasy RBTree 1482
36.7. Testowanie klasy RBTree 1489
36.8. Wydajność klasy RBTree 1492

Rozdział 37. Testy z użyciem JUnit 1497

37.1. Wprowadzenie 1498
37.2. Podstawy JUnit 1498
37.3. Używanie JUnit w NetBeans 1504
37.4. Używanie JUnit w Eclipse 1507

Dodatek A Słowa kluczowe i zarezerwowane w Javie 1513

Dodatek B Zestaw znaków ASCII 1515

Dodatek C Tabela priorytetów operatorów 1517

Dodatek D Modyfikatory w Javie 1519

Dodatek E Specjalne wartości zmiennoprzecinkowe 1521

Dodatek F Systemy liczbowe 1523

Dodatek G Operacje bitowe 1527

Dodatek H Wyrażenia regularne 1529

Dodatek I Typy wyliczeniowe 1537

Dodatek J Notacje dużego O, dużego omega i dużego theta 1543

Kup ksi k Pole ksi k

ROZDZIAŁ

23

SORTOWANIE

Cele

- Zbadanie i przeanalizowanie złożoności czasowej różnych algorytmów sortowania (podrozdziały 23.2 – 23.7).
- Zaprojektowanie, zaimplementowanie i przeanalizowanie sortowania przez wstawianie (podrozdział 23.2).
- Zaprojektowanie, zaimplementowanie i przeanalizowanie sortowania bąbelkowego (podrozdział 23.3).
- Zaprojektowanie, zaimplementowanie i przeanalizowanie sortowania przez scalanie (podrozdział 23.4).
- Zaprojektowanie, zaimplementowanie i przeanalizowanie sortowania szybkiego (podrozdział 23.5).
- Zaprojektowanie i zaimplementowanie kopca binarnego (podrozdział 23.6).
- Zaprojektowanie, zaimplementowanie i przeanalizowanie sortowania przez kopcowanie (podrozdział 23.6).
- Zaprojektowanie, zaimplementowanie i przeanalizowanie sortowania kubełkowego oraz sortowania pozycyjnego (podrozdział 23.7).
- Zaprojektowanie, zaimplementowanie i przeanalizowanie zewnętrznego sortowania plików zawierających duże ilości danych (podrozdział 23.8).

Kup książkę Poleć książkę

23.1. Wprowadzenie

Algorytmy sortowania dobrze nadają się do badania projektowania i analiz algorytmów.

Gdy prezydent Barack Obama odwiedził w 2007 r. firmę Google, CEO firmy, Eric Schmidt, zapytał prezydenta o najwydajniejszy sposób sortowania miliona 32-bitowych liczb całkowitych (*http://www.youtube.com/watch?v=k4RRi_ntQc8*). Obama odpowiedział, że sortowanie bąbelkowe byłoby złym pomysłem. Czy miał rację? W tym rozdziale poznasz różne algorytmy sortowania i dowiesz się, czy odpowiedź Obamy była poprawna.

po co badać sortowanie?

Sortowanie jest klasycznym zagadnieniem w dziedzinie informatyki. Są trzy powody, dla których warto zbadać algorytmy sortowania.

- Po pierwsze, algorytmy sortowania ilustrują wiele pomysłowych technik rozwiązywania problemów. Techniki te można wykorzystać także do innych zadań.
- Po drugie, algorytmy sortowania dobrze nadają się do przećwiczenia podstawowych technik programowania z wykorzystaniem instrukcji sterujących, pętli, metod i tablic.
- Po trzecie, algorytmy sortowania doskonale nadają się do demonstrowania wydajności algorytmów.

jakie dane można sortować?

Sortowanymi danymi mogą być liczby całkowite, liczby zmiennoprzecinkowe, znaki lub obiekty. W podrozdziale 7.11, „Sortowanie tablic”, opisano sortowanie przez wybieranie. W podrozdziale 19.5, „Studium przypadku: sortowanie tablicy obiektów”, algorytm sortowania przez wybieranie został rozbudowany o sortowanie tablic obiektów. API Javy zawiera w klasach `java.util.Arrays` i `java.util.Collections` kilka przeciążonych metod do sortowania wartości typów podstawowych i obiektów. Dla uproszczenia w tym rozdziale przyjęto, że:

1) sortowanymi danymi są liczby całkowite,
2) dane są zapisane w tablicy,
3) dane są sortowane rosnąco.

Przedstawione programy można łatwo zmodyfikować, aby sortować inne typy danych, sortować dane malejąco lub sortować dane z listy `ArrayList` i `LinkedList`.

Istnieje wiele algorytmów sortowania. Poznałeś już sortowanie przez wybieranie. W tym rozdziale dowiesz się, jak działa sortowanie przez wstawianie, sortowanie bąbelkowe, sortowanie przez scalanie, sortowanie szybkie, sortowanie kubełkowe, sortowanie pozycyjne i sortowanie zewnętrzne.

23.2. Sortowanie przez wstawianie

Algorytm sortowania przez wstawianie sortuje listę wartości, wielokrotnie wstawiając nowy element na posortowaną podlistę do czasu posortowania całej listy.

animacja sortowania przez wstawianie w witrynie

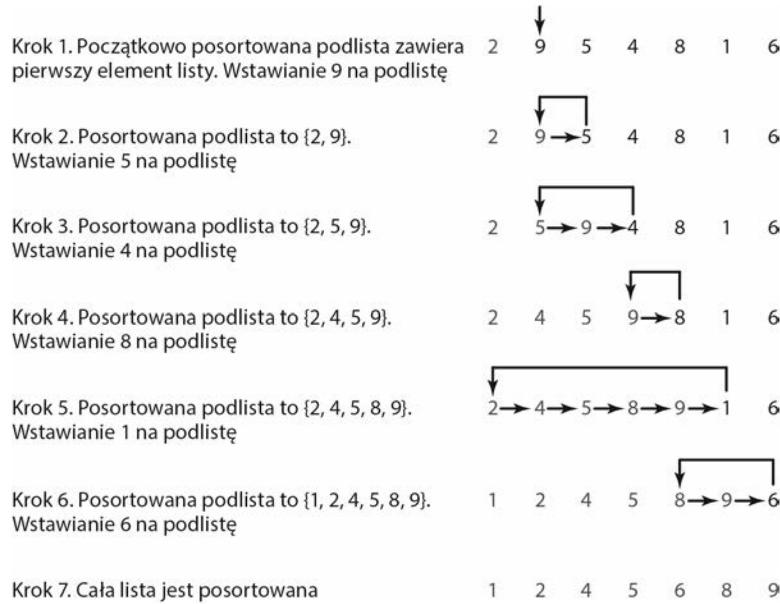
Na rysunku 23.1 pokazane jest, jak posortować listę {2, 9, 5, 4, 8, 1, 6} za pomocą sortowania przez wstawianie. Interaktywną ilustrację sortowania przez wstawianie znajdziesz na stronie *http://liveexample.pearsoncmg.com/dsanimation/InsertionSortNeweBook.html.*

Ten algorytm można opisać tak:

```
for (int i = 1; i < list.length; i++) {
  wstaw list[i] na posortowaną podlistę list[0..i-1] w taki sposób, aby
  elementy list[0..i] były posortowane.
}
```

Aby wstawić `list[i]` na listę `list[0..i-1]`, zapisz `list[i]` w zmiennej pomocniczej (na przykład `currentElement`). Przenieś `list[i-1]` do `list[i]`, jeśli `list[i-1] > currentElement`, `list[i-2]` do `list[i-1]`, jeśli

Kup ksi k Pole ksi k



RYSUNEK 23.1. Sortowanie przez wstawianie wymaga wielokrotnego wstawiania nowego elementu na posortowaną podlistę

`list[i-2] > currentElement` itd. — do czasu, gdy `list[i-k] <= currentElement` lub `k > i` (algorytm wychodzi poza pierwszy element posortowanej listy). Przypisz `currentElement` do `list[i-k+1]`. Na przykład aby wstawić 4 na listę {2, 5, 9} w kroku 4. z rysunku 23.2, przenieś `list[2]` (czyli 9) do `list[3]`, ponieważ 9 > 4, i przenieś `list[1]` (czyli 5) do `list[2]`, ponieważ 5 > 4. W ostatnim kroku przenieś `currentElement` (4) do `list[1]`.

```
     [0] [1] [2] [3] [4] [5] [6]
list  2   5   9   4                 Krok 1. Zapisz 4 w zmiennej pomocniczej currentElement
                                    currentElement: 4

     [0] [1] [2] [3] [4] [5] [6]
list  2   5       9                 Krok 2. Przenieś list[2] do list[3]

     [0] [1] [2] [3] [4] [5] [6]
list  2       5   9                 Krok 3. Przenieś list[1] do list[2]

     [0] [1] [2] [3] [4] [5] [6]
list  2   4   5   9                 Krok 4. Przenieś currentElement do list[1]
```

RYSUNEK 23.2. Nowy element jest wstawiany na posortowaną podlistę

Ten algorytm można rozwinąć i zaimplementować w sposób pokazany na listingu 23.1.

Kup ksi k Pole ksi k

LISTING 23.1. InsertionSort.java

```
 1 public class InsertionSort {
 2   /** Metoda do sortowania liczb */
 3   public static void insertionSort(int[] list) {
 4     for (int i = 1; i < list.length; i++) {
 5       /** Wstawianie list[i] na posortowaną podlistę list[0..i–1], aby
 6           fragment list[0..i] był posortowany. */
 7       int currentElement = list[i];
 8       int k;
 9       for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
10         list[k + 1] = list[k];
11       }
12
13       // Przypisywanie bieżącego elementu do list[k + 1]
14       list[k + 1] = currentElement;
15     }
16   }
17 }
```

przesunięcie

wstawianie

Metoda `insertionSort(int[] list)` sortuje tablicę elementów typu `int`. Ta metoda jest zaimplementowana z użyciem zagnieżdżonej pętli `for`. Pętla zewnętrzna (ze zmienną sterującą `i`; wiersz 4.) powtarza iteracje, aby utworzyć posortowaną podlistę od `list[0]` do `list[i]`. Pętla wewnętrzna (ze zmienną sterującą `k`) wstawia `list[i]` na podlistę od `list[0]` do `list[i-1]`.

Aby lepiej zrozumieć tę metodę, prześledź jej działanie dla następujących instrukcji:

```
int[] list = {1, 9, 4, 6, 5, -4};
InsertionSort.insertionSort(list);
```

złożoność czasowa sortowania przez wstawianie

Zaprezentowany tu algorytm sortowania przez wstawianie sortuje listę elementów, wielokrotnie wstawiając nowy element do posortowanego fragmentu tablicy do czasu posortowania jej całej. W k-tej iteracji wstawianie elementu do tablicy o wielkości k może wymagać k porównań (aby znaleźć pozycję wstawiania) i k przesunięć (aby wstawić element). Niech $T(n)$ oznacza złożoność sortowania przez wstawianie, a c — łączną liczbę innych operacji takich jak przypisania i dodatkowe porównania w każdej iteracji:

$$
\begin{aligned}
T(n) &= (2+c) + (2\times 2 + c) + \ldots + (2\times(n-1) + c) \\
&= (2(1+2+\ldots+n-1) + c(n-1) \\
&= 2\frac{(n-1)n}{2} + cn - c = n^2 - n + cn - c \\
&= O(n^2)
\end{aligned}
$$

Tak więc złożoność algorytmu sortowania przez wstawianie wynosi $O(n^2)$. Oznacza to, że złożoność czasowa sortowania przez wybieranie i sortowania przez wstawianie jest taka sama.

23.2.1. Opisz działanie sortowania przez wstawianie. Jaka jest złożoność czasowa tego algorytmu?

23.2.2. Na podstawie rysunku 23.1 przedstaw przebieg sortowania przez wstawianie listy {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.2.3. Ile porównań wykona metoda `insertionSort`, jeśli lista jest już posortowana?

Kup ksi k   Pole ksi k

23.3. Sortowanie bąbelkowe

Sortowanie bąbelkowe sortuje tablicę w wielu przebiegach. W każdym z nich sąsiednie elementy są ze sobą przestawiane, jeśli ich kolejność jest nieprawidłowa.

sortowanie bąbelkowe

Algorytm sortowania bąbelkowego wykonuje kilka przebiegów przez tablicę. W każdym przebiegu porównywane są kolejne sąsiadujące pary. Jeśli elementy są uporządkowane w kolejności malejącej, należy je przestawić. W przeciwnym razie elementy pozostają bez zmian. Ta technika jest nazywana *sortowaniem bąbelkowym*, ponieważ mniejsze wartości stopniowo „wypływają" na powierzchnię, a większe wartości „opadają" na dno. Po pierwszym przebiegu największy element zajmuje ostatnią pozycję w tablicy. Po drugim przebiegu drugi największy element trafia na przedostatnią pozycję. Proces ten jest kontynuowany do czasu posortowania wszystkich elementów.

ilustracja sortowania bąbelkowego

animacja sortowania bąbelkowego w witrynie

Na rysunku 23.3a pokazany jest pierwszy przebieg sortowania bąbelkowego tablicy sześciu elementów (2 9 5 4 8 1). Elementy pierwszej pary (2 i 9) nie wymagają przestawiania, ponieważ są uporządkowane. Elementy drugiej pary (9 i 5) wymagają przestawienia, ponieważ 9 jest większe niż 5. Elementy trzeciej pary (9 i 4) wymagają przestawienia 9 z 4. Elementy czwartej pary (9 i 8) wymagają przestawienia 9 z 8. Elementy piątej pary (9 i 1) wymagają przestawienia 9 z 1. Na rysunku 23.3 porównywane pary są wyróżnione, a już posortowane wartości są wyróżnione kursywą. Interaktywną ilustrację sortowania bąbelkowego znajdziesz na stronie *http://liveexample.pearsoncmg.com/dsanimation/BubbleSortNeweBook.html*.

(a) Pierwszy przebieg:
2 9 5 4 8 1
2 5 9 4 8 1
2 5 4 9 8 1
2 5 4 8 9 1
2 5 4 8 1 9

(b) Drugi przebieg:
2 5 4 8 1 9
2 4 5 8 1 9
2 4 5 8 1 9
2 4 5 1 8 9

(c) Trzeci przebieg:
2 4 5 1 8 9
2 4 5 1 8 9
2 4 1 5 8 9

(d) Czwarty przebieg:
2 4 1 5 8 9
2 1 4 5 8 9

(e) Piąty przebieg:
1 2 4 5 8 9

RYSUNEK 23.3. W każdym przebiegu algorytm po kolei porównuje i porządkuje pary

W pierwszym przebiegu największa wartość (9) jest umieszczana na końcu tablicy. W drugim przebiegu (rysunek 23.3b) algorytm po kolei porównuje i porządkuje pary. Ostatniej pary nie trzeba sprawdzać, ponieważ ostatni element tablicy jest już największy. W trzecim przebiegu (rysunek 23.3c) algorytm porównuje i porządkuje pary oprócz dwóch ostatnich, ponieważ dwa końcowe elementy są już uporządkowane. W k-tym przebiegu nie trzeba uwzględniać ostatnich $k - 1$ elementów, ponieważ są już uporządkowane.

algorytm

Algorytm sortowania bąbelkowego jest pokazany na listingu 23.2.

LISTING 23.2. Algorytm sortowania bąbelkowego

```
1 for (int k = 1; k < list.length; k++) {
2   // K-ty przebieg
3   for (int i = 0; i < list.length - k; i++) {
4     if (list[i] > list[i + 1])
5       swap list[i] with list[i + 1];
6   }
7 }
```

Kup ksi k Pole ksi k

Zauważ, że jeśli w danym przebiegu nie przestawiono żadnych elementów, dalsze przebiegi są zbędne, ponieważ wszystkie elementy zostały już posortowane. Możesz wykorzystać to spostrzeżenie, aby usprawnić algorytm z listingu 23.2 w sposób pokazany na listingu 23.3.

LISTING 23.3. Ulepszony algorytm sortowania bąbelkowego

```
 1 boolean needNextPass = true;
 2 for (int k = 1; k < list.length && needNextPass; k++) {
 3   // Możliwe, że tablica jest posortowana i następny przebieg nie jest potrzebny
 4   needNextPass = false;
 5   // Wykonywanie k-tego przebiegu
 6   for (int i = 0; i < list.length – k; i++) {
 7     if (list[i] > list[i + 1]) {
 8       swap list[i] with list[i + 1];
 9       needNextPass = true; // Następny przebieg jest potrzebny
10     }
11   }
12 }
```

Implementację tego algorytmu przedstawia listing 23.4.

Listing 23.4. BubbleSort.java

wykonanie jednego przebiegu

```
 1 public class BubbleSort {
 2   /** Metoda wykonująca sortowanie bąbelkowe */
 3   public static void bubbleSort(int[] list) {
 4     boolean needNextPass = true;
 5
 6     for (int k = 1; k < list.length && needNextPass; k++) {
 7       // Możliwe, że tablica jest posortowana i następny przebieg nie jest potrzebny
 8       needNextPass = false;
 9       for (int i = 0; i < list.length - k; i++) {
10         if (list[i] > list[i + 1]) {
11           // Przestawianie list[i] z list[i + 1]
12           int temp = list[i];
13           list[i] = list[i + 1];
14           list[i + 1] = temp;
15
16           needNextPass = true; // Następny przebieg jest potrzebny
17         }
18       }
19     }
20   }
21
22   /** Metoda testowa */
23   public static void main(String[] args) {
24     int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
25     bubbleSort(list);
26     for (int i = 0; i < list.length; i++)
27       System.out.print(list[i] + " ");
28   }
29 }
```

```
-2 1 2 2 3 3 5 6 12 14
```

W optymistycznym przypadku algorytm sortowania bąbelkowego wymaga tylko jednego przebiegu, aby wykryć, że tablica jest już posortowana. Dalsze przebiegi nie są potrzebne. Ponieważ liczba porównań w pierwszym przebiegu to $n - 1$, w optymistycznym przypadku sortowanie bąbelkowe ma złożoność $O(n)$.

złożoność czasowa sortowania bąbelkowego

W pesymistycznym przypadku algorytm sortowania bąbelkowego wymaga $n - 1$ przebiegów. W pierwszym wykonywanych jest $n - 1$ porównań, w drugim — $n - 2$ porównań itd. Ostatni przebieg wymaga jednego porównania. Łączna liczba porównań wynosi więc:

$$(n-1)+(n-2)+...+2+1$$
$$=\frac{(n-1)n}{2}=\frac{n^2}{2}-\frac{n}{2}=O(n^2)$$

Dlatego w pesymistycznym przypadku sortowanie bąbelkowe ma złożoność $O(n^2)$.

23.3.1. Opisz działanie sortowania bąbelkowego. Jaka jest złożoność czasowa tego algorytmu?

23.3.2. Wzorując się na rysunku 23.3, pokaż przebieg sortowania bąbelkowego liczb {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.3.3. Ile porównań musi wykonać metoda `bubbleSort`, jeśli lista jest już posortowana?

23.4. Sortowanie przez scalanie

Algorytm sortowania przez scalanie można opisać rekurencyjnie: podziel tablicę na dwie połowy i zastosuj rekurencyjnie sortowanie przez scalanie do każdej z nich; po posortowaniu obu połów scal je.

sortowanie przez scalanie

Algorytm *sortowania przez scalanie* jest opisany na listingu 23.5.

LISTING 23.5. Algorytm sortowania przez scalanie

przypadek bazowy
sortowanie pierwszej połowy
sortowanie drugiej połowy
scalanie obu połów

```
1 public static void mergeSort(int[] list) {
2   if (list.length > 1) {
3     mergeSort(list[0 ... list.length / 2]);
4     mergeSort(list[list.length / 2 + 1 ... list.length]);
5     scal list[0 ... list.length / 2] z
6     list[list.length / 2 + 1 ... list.length];
7   }
8 }
```

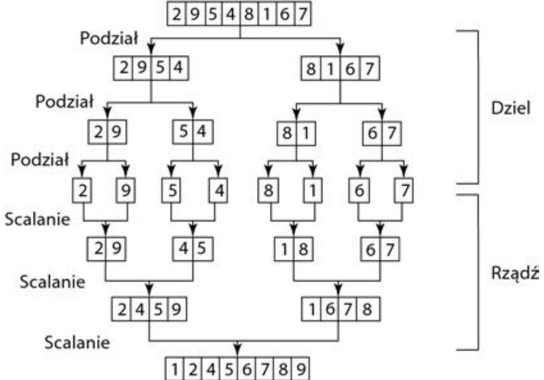
ilustracja sortowania przez scalanie

Na rysunku 23.4 pokazane jest sortowanie przez scalanie tablicy ośmiu elementów (2 9 5 4 8 1 6 7). Pierwotna tablica jest dzielona na fragmenty (2 9 5 4) i (8 1 6 7). Należy rekurencyjnie przeprowadzić sortowanie przez scalanie tych dwóch podtablic, dzieląc (2 9 5 4) na (2 9) i (5 4) oraz (8 1 6 7) na (8 1) i (6 7). Ten proces jest kontynuowany do momentu, gdy podtablica zawiera tylko jeden element. Na przykład tablica (2 9) jest dzielona na podtablice (2) i (9). Ponieważ tablica (2) zawiera jeden element, nie da się jej dalej podzielić. Teraz należy scalić (2) i (9) w nową posortowaną tablicę (2 9), a (5) i (4) w nową posortowaną tablicę (4 5). Następnie scal (2 9) i (4 5) w nową posortowaną tablicę (2 4 5 9), a w ostatnim kroku scal (2 4 5 9) i (1 6 7 8) w nową posortowaną tablicę (1 2 4 5 6 7 8 9).

Wywołania rekurencyjne dzielą tablicę na podtablice do momentu, gdy każda podtablica zawiera tylko jeden element. Następnie algorytm scala małe podtablice w większe posortowane podtablice do momentu uzyskania jednej posortowanej tablicy.

Implementacja algorytmu sortowania przez scalanie jest pokazana na listingu 23.6.

Kup ksi k Pole ksi k



RYSUNEK 23.4. W sortowaniu przez scalanie tablica jest sortowana metodą dziel i rządź

LISTING 23.6. MergeSort.java

```
 1 public class MergeSort {
 2   /** Metoda do sortowania liczb */
 3   public static void mergeSort(int[] list) {
 4     if (list.length > 1) {
 5       // Sortowanie przez scalanie pierwszej połowy
 6       int[] firstHalf = new int[list.length / 2];
 7       System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
 8       mergeSort(firstHalf);
 9
10       // Sortowanie przez scalanie drugiej połowy
11       int secondHalfLength = list.length - list.length / 2;
12       int[] secondHalf = new int[secondHalfLength];
13       System.arraycopy(list, list.length / 2,
14         secondHalf, 0, secondHalfLength);
15       mergeSort(secondHalf);
16
17       // Scalanie firstHalf z secondHalf w list
18       merge(firstHalf, secondHalf, list);
19     }
20   }
21
22   /** Scalanie dwóch posortowanych list */
23   public static void merge(int[] list1, int[] list2, int[] temp) {
24     int current1 = 0; // Indeks bieżącego elementu w list1
25     int current2 = 0; // Indeks bieżącego elementu w list2
26     int current3 = 0; // Indeks bieżącego elementu w temp
27
28     while(current1 < list1.length && current2 < list2.length) {
29       if (list1[current1] < list2[current2])
30         temp[current3++] = list1[current1++];
31       else
32         temp[current3++] = list2[current2++];
33     }
34
35     while (current1 < list1.length)
36       temp[current3++] = list1[current1++];
```

Uwagi na marginesie: przypadek bazowy (wiersz 4); sortowanie pierwszej połowy (wiersz 8); sortowanie drugiej połowy (wiersz 15); scalanie obu połów (wiersz 18); z list1 do temp (wiersz 30); z list2 do temp (wiersz 32); reszta z list1 do temp (wiersz 35).

Kup książkę Poleć książkę

reszta z list2 do temp

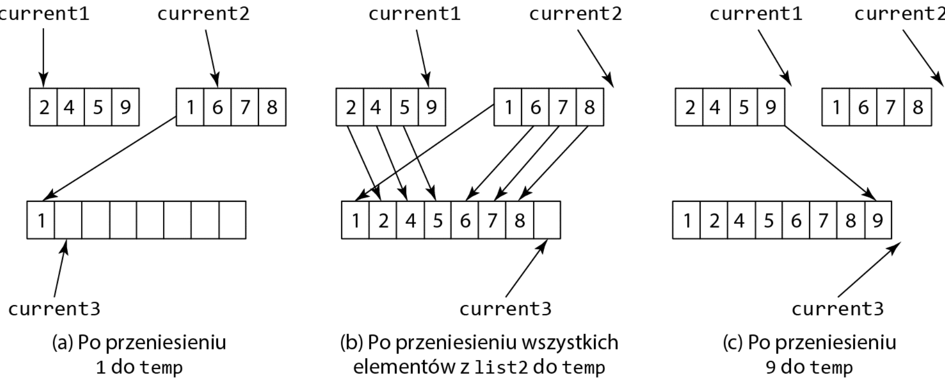
```
37
38     while (current2 < list2.length)
39       temp[current3++] = list2[current2++];
40   }
41
42   /** Metoda testowa */
43   public static void main(String[] args) {
44     int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
45     mergeSort(list);
46     for (int i = 0; i < list.length; i++)
47       System.out.print(list[i] + " ");
48   }
49 }
```

Metoda `mergeSort` (wiersze 3. – 20.) tworzy nową tablicę `firstHalf`, która jest kopią pierwszej połowy tablicy `list` (wiersz 7.). Algorytm rekurencyjnie wywołuje metodę `mergeSort` dla tablicy `firstHalf` (wiersz 8.). Długość tablicy `firstHalf` wynosi `list.length / 2`, a długość tablicy `secondHalf` to `list.length - list.length / 2`. Nowa tablica `secondHalf` zawiera drugą część pierwotnej tablicy `list`. Algorytm wywołuje metodę `mergeSort` rekurencyjnie dla tablicy `secondHalf` (wiersz 15.). Po posortowaniu tablic `firstHalf` i `secondHalf` kod scala je w tablicę `list` (wiersz 18.). Tablica `list` jest teraz posortowana.

Metoda `merge` (wiersze 23. – 40.) scala dwie posortowane tablice `list1` i `list2` w tablicę `temp`. `current1` i `current2` wskazują analizowane elementy z `list1` i `list2` (wiersze 24. – 26.). Metoda wielokrotnie porównuje bieżące elementy z `list1` i `list2` oraz przenosi mniejszy z nich do tablicy `temp`. Jeśli mniejszy element znajduje się w `list1`, kod zwiększa wartość `current1` o 1 (wiersz 30.). Jeżeli mniejszy element pochodzi z `list2`, algorytm zwiększa wartość `current2` o 1 (wiersz 32.). Ostatecznie wszystkie elementy z jednej z list zostają przeniesione do `temp`. Jeśli w `list1` znajdują się jeszcze jakieś elementy, należy je skopiować do `temp` (wiersze 35. i 36.). Jeżeli w `list2` występują nieprzeniesione elementy, kod kopiuje je do `temp` (wiersze 38. i 39.).

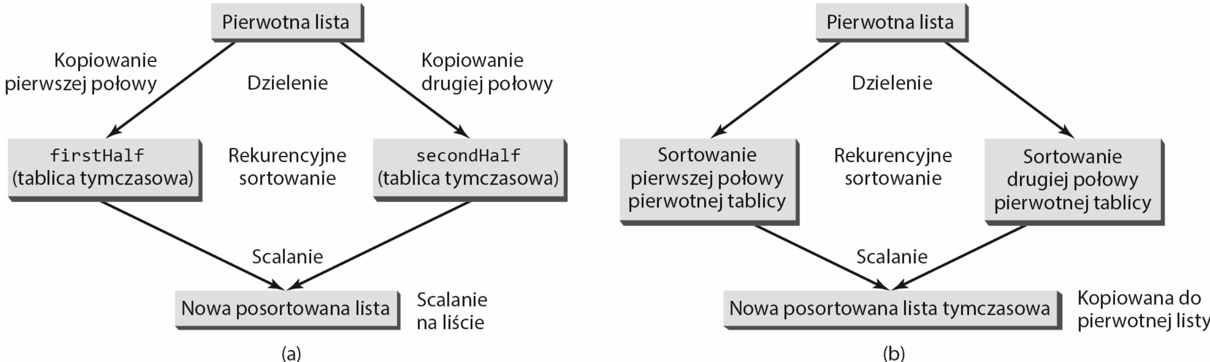
Na rysunku 23.5 pokazane jest, jak scalić dwie tablice, `list1` (2 4 5 9) i `list2` (1 6 7 8). Początkowe bieżące elementy sprawdzane w tablicach to 2 i 1. Po porównaniu mniejsza wartość (1) jest przenoszona do `temp` (rysunek 23.5a), a `current2` i `current3` są zwiększane o 1. Należy porównywać kolejne bieżące elementy z obu tablic i przenosić mniejszy z nich do `temp` do czasu całkowitego przeniesienia jednej z nich. Na rysunku 23.5b wszystkie elementy z `list2` zostały już przeniesione do `temp`, a `current1` wskazuje element 9 z `list1`. Trzeba więc skopiować 9 do `temp` (rysunek 23.5c). Interaktywną ilustrację scalania znajdziesz na stronie *http://liveexample.pearsoncmg.com/dsanimation/MergeSortNeweBook.html*.

animacja scalania w witrynie



RYSUNEK 23.5. Dwie posortowane tablice są scalane w jedną

Metoda `mergeSort` w procesie podziału tworzy dwie tablice tymczasowe (wiersze 6. i 12.), kopiuje pierwszą i drugą połowę do tymczasowych tablic (wiersze 7. i 13.), sortuje tymczasowe tablice (wiersze 8. i 15.), a następnie scala je (wiersz 18.). Ilustruje to rysunek 23.6a. Możesz zmodyfikować kod, aby rekurencyjnie sortować pierwszą i drugą połowę tablicy bez tworzenia nowych tablic tymczasowych, a następnie scalać obie tablice w tymczasową i kopiować jej zawartość do pierwotnej tablicy (rysunek 23.6b). Potraktuj to jako ćwiczenie (ćwiczenie 23.20).



RYSUNEK 23.6. Na potrzeby sortowania przez scalanie tworzone są tablice tymczasowe

Uwaga

Sortowanie przez scalanie można wydajnie zaimplementować z wykorzystaniem przetwarzania równoległego. Taką implementację znajdziesz w podrozdziale 32.16, „Programowanie równoległe”.

złożoność czasowa sortowania przez scalanie

Przeanalizuj teraz czas sortowania przez scalanie. Niech $T(n)$ oznacza czas sortowania tablicy n elementów. Bez utraty ogólności można przyjąć, że n jest potęgą liczby 2. Algorytm sortowania przez scalanie dzieli tablicę na dwie podtablice, sortuje podtablice rekurencyjnie za pomocą tego samego algorytmu, a następnie scala podtablice. Mamy więc:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + czas_scalania$$

Pierwsze $T(\frac{n}{2})$ to czas sortowania pierwszej połowy tablicy; drugie $T(\frac{n}{2})$ to czas sortowania drugiej połowy. Aby scalić podtablice, potrzeba najwyżej $n - 1$ porównań elementów z obu podtablic i n przeniesień elementów do tablicy tymczasowej. Łączny czas wynosi więc $2n - 1$. Otrzymujemy zatem:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 2n - 1 = O(n \log n)$$

sortowanie przez scalanie działa w czasie O(n logn)

Złożoność sortowania przez scalanie to $O(n \log n)$. Algorytm jest szybszy od sortowania przez wybieranie, sortowania przez scalanie i sortowania bąbelkowego, których złożoność to $O(n^2)$. Metoda `sort` z klasy `java.util.Arrays` jest zaimplementowana za pomocą zmodyfikowanego sortowania przez scalanie.

23.4.1. Opisz sortowanie przez scalanie. Jaka jest złożoność czasowa tego algorytmu?

23.4.2. Wzorując się na rysunku 23.4, pokaż przebieg sortowania przez scalanie dla listy {45, 11, 50, 59, 60, 2, 4, 7, 10}.

Kup ksi k Pole ksi k

23.4.3. Jaki błąd się pojawi, jeśli wiersze 6. – 15. z listingu 23.6, *MergeSort.java*, zostaną zastąpione poniższym kodem?

```
// Sortowanie przez scalanie pierwszej połowy
int[] firstHalf = new int[list.length / 2 + 1];
System.arraycopy(list, 0, firstHalf, 0, list.length / 2 + 1);
mergeSort(firstHalf);

// Sortowanie przez scalanie drugiej połowy
int secondHalfLength = list.length - list.length / 2 - 1;
int[] secondHalf = new int[secondHalfLength];
System.arraycopy(list, list.length / 2 + 1,
  secondHalf, 0, secondHalfLength);
mergeSort(secondHalf);
```

23.5. Sortowanie szybkie

Sortowanie szybkie działa tak: algorytm wybiera w tablicy element osiowy. Ten element dzieli tablicę na dwie części, gdzie w pierwszej części znajdują się elementy mniejsze od osiowego lub mu równe, a w drugiej — większe od osiowego. Algorytm jest następnie rekurencyjnie stosowany do obu części.

sortowanie szybkie

Algorytm sortowania szybkiego, opracowany przez C.A.R Hoare a w 1962 r., jest opisany na listingu 23.7.

Listing 23.7. Algorytm sortowania szybkiego

przypadek bazowy; wybór elementu osiowego; podział listy; sortowanie pierwszej części; sortowanie drugiej części

```
 1 public static void quickSort(int[] list) {
 2   if (list.length > 1) {
 3     wybór elementu osiowego;
 4     podział listy na list1 i list2, aby
 5       wszystkie elementy w list1 <= osiowy i
 6       wszystkie elementy w list2 > osiowy;
 7     quickSort(list1);
 8     quickSort(list2);
 9   }
10 }
```

jak podzielić listę?

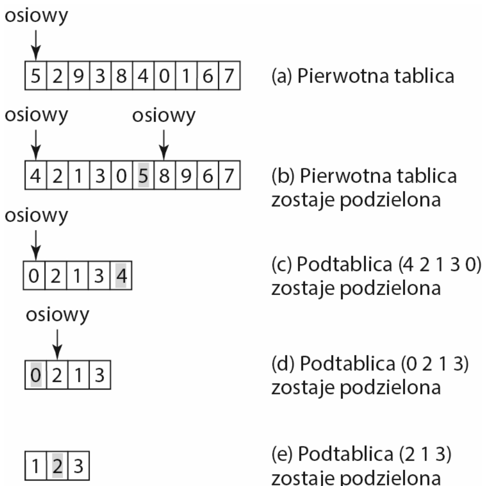
Każdy podział powoduje umieszczenie elementu osiowego w odpowiednim miejscu. Lista jest dzielona na dwie podlisty w następujący sposób:

Wydajność algorytmu zależy od wyboru elementu osiowego. W idealnym scenariuszu element osiowy powinien dzielić listę na dwie równe części. Dla uproszczenia przyjmij, że jako element osiowy wybierany jest pierwszy element tablicy (w ćwiczeniu 23.4 zaproponowana jest inna strategia).

ilustracja sortowania szybkiego

Na rysunku 23.7 pokazano, jak posortować tablicę (5 2 9 3 8 4 0 1 6 7) za pomocą sortowania szybkiego. Jako osiowy wybierany jest pierwszy element, 5. Tablica jest dzielona na dwie części (rysunek 23.7b). Wyróżniony element osiowy jest umieszczany w odpowiednim miejscu, a sortowanie szybkie jest stosowane do dwóch podtablic: (4 2 1 3 0) i (8 9 6 7). Element osiowy 4 dzieli (4 2 1 3 0) na tylko jedną podtablicę (0 2 1 3) (rysunek 23.7c). Przy sortowaniu szybkim tablica (0 2 1 3) jest dzielona przez 0 na jedną podtablicę (2 1 3) (rysunek 23.7d). Należy zastosować sortowanie szybkie do tablicy (2 1 3). Element osiowy 2 dzieli ją na (1) i (3) (rysunek 23.7e). Sortowanie szybkie jest stosowane do podtablicy (1). Ponieważ zawiera ona tylko jeden element, dalsze podziały nie są potrzebne.

Kup ksi k Pole ksi k



RYSUNEK 23.7. Algorytm szybkiego sortowania jest rekurencyjnie stosowany do podtablic

Implementację algorytmu szybkiego sortowania przedstawia listing 23.8. W tej klasie znajdują się dwie przeciążone metody `quickSort`. Pierwsza z nich (wiersz 2.) sortuje tablicę. Druga to metoda pomocnicza (wiersz 6.), która sortuje podtablicę o określonym zakresie.

LISTING 23.8. QuickSort.java

metoda sortująca
metoda pomocnicza
wywołanie rekurencyjne
do przodu
wstecz

```
 1 public class QuickSort {
 2   public static void quickSort(int[] list) {
 3     quickSort(list, 0, list.length - 1);
 4   }
 5
 6   public static void quickSort(int[] list, int first, int last) {
 7     if (last > first) {
 8       int pivotIndex = partition(list, first, last);
 9       quickSort(list, first, pivotIndex - 1);
10       quickSort(list, pivotIndex + 1, last);
11     }
12   }
13
14   /** Podział tablicy list[first..last] */
15   public static int partition(int[] list, int first, int last) {
16     int pivot = list[first]; // Wybór pierwszego elementu jako osiowego
17     int low = first + 1; // Indeks do wyszukiwania do przodu
18     int high = last; // Indeks do wyszukiwania wstecz
19
20     while (high > low) {
21       // Przeszukiwanie od lewej do przodu
22       while (low <= high && list[low] <= pivot)
23         low++;
24
25       // Przeszukiwanie od prawej wstecz
26       while (low <= high && list[high] > pivot)
27         high--;
```

Kup książkę Poleć książkę

przestawianie

umieszczanie elementu osiowego

nowy indeks elementu osiowego

pierwotny indeks elementu osiowego

```
28
29         // Przestawianie dwóch elementów listy
30         if (high > low) {
31           int temp = list[high];
32           list[high] = list[low];
33           list[low] = temp;
34         }
35       }
36
37       while (high > first && list[high] >= pivot)
38         high--;
39
40       // Przestawianie elementu osiowego z list[high]
41       if (pivot > list[high]) {
42         list[first] = list[high];
43         list[high] = pivot;
44         return high;
45       }
46       else {
47         return first;
48       }
49     }
50
51     /** Metoda testowa */
52     public static void main(String[] args) {
53       int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
54       quickSort(list);
55       for (int i = 0; i < list.length; i++)
56         System.out.print(list[i] + " ");
57     }
58 }
```

```
-2 1 2 2 3 3 5 6 12 14
```

Metoda `partition` (wiersze 15. – 49.) dzieli tablicę `list[first..last]` za pomocą elementu osiowego. Jako osiowy jest wybierany pierwszy element części tablicy (wiersz 16.). Początkowo `low` wskazuje drugi element podtablicy (wiersz 17.), a `high` — ostatni element podtablicy (wiersz 18.).

Począwszy od lewej, metoda przeszukuje tablicę do przodu, aby znaleźć w niej pierwszy element większy od osiowego (wiersze 22. i 23.). Następnie metoda przeszukuje tablicę od prawej wstecz, szukając pierwszego elementu mniejszego lub równego względem osiowego (wiersze 26. i 27.). Potem tablica przestawia te dwa elementy oraz powtarza wyszukiwanie i przestawianie do czasu sprawdzenia w pętli `while` wszystkich elementów (wiersze 20. – 35.).

Jeśli element osiowy został przestawiony, metoda zwraca jego nowy indeks (wiersz 44.). W przeciwnym razie zwracany jest pierwotny indeks elementu osiowego (wiersz 47.).

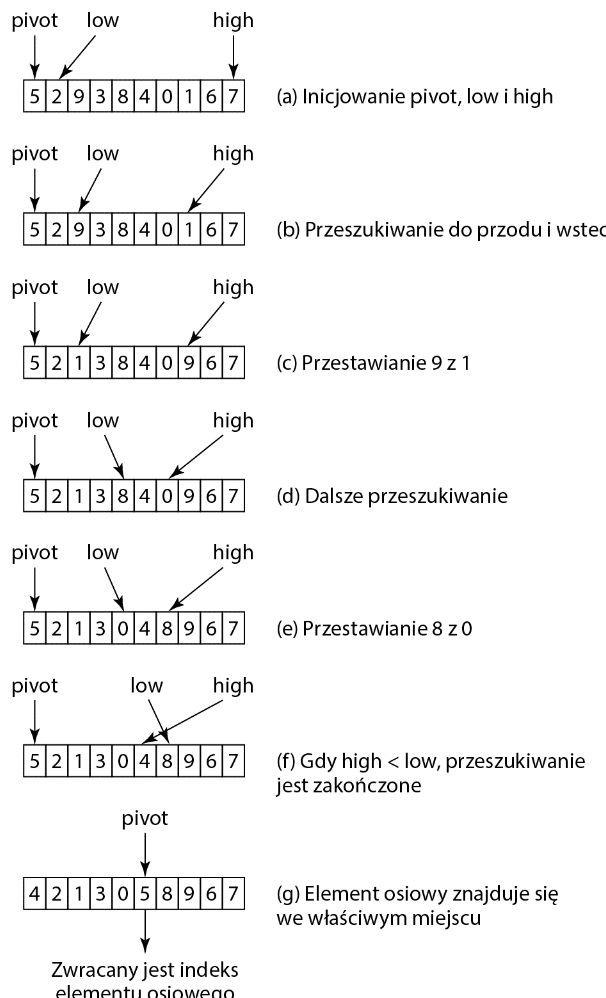
ilustracja podziału

Na rysunku 23.8 pokazany jest podział tablicy (5 2 9 3 8 4 0 1 6 7). Jako osiowy (`pivot`) wybierany jest pierwszy element, 5. Początkowo `low` to indeks elementu 2, a `high` to indeks elementu 7 (rysunek 23.8a). Indeks `low` jest zwiększany, aby znaleźć pierwszy element (9) większy od osiowego, a indeks `high` jest zmniejszany w poszukiwaniu pierwszego elementu (1) mniejszego od osiowego lub mu równego (rysunek 23.8b). Należy przestawić 9 z 1 (rysunek 23.8c) i kontynuować wyszukiwanie. Na rysunku 23.8d indeks `low` wskazuje element 8, a indeks `high` — element 0, dlatego należy przestawić 8 z 0 (rysunek 23.8e). Proces jest kontynuowany do czasu, gdy `low` staje się

Kup ksi k Pole ksi k

animacja podziału w witrynie

większe od `high` (rysunek 23.8f). Oznacza to, że wszystkie elementy zostały sprawdzone. Element osiowy należy przestawić z elementem 4 (o indeksie `high`). Ostateczny podział jest pokazany na rysunku 23.8g. Po zakończeniu pracy metoda zwraca indeks elementu osiowego. Interaktywną ilustrację podziału znajdziesz na stronie *http://liveexample.pearsoncmg.com/dsanimation/QuickSortNeweBook.html*.



RYSUNEK 23.8. Metoda partition zwraca indeks elementu osiowego po jego umieszczeniu we właściwym miejscu

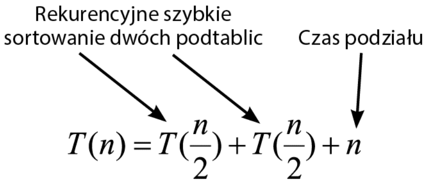
złożoność podziału to O(n)

Podział tablicy n elementów w pesymistycznym przypadku wymaga n porównań i n przeniesień. Czas podziału wynosi więc $O(n)$.

złożoność dla przypadku pesymistycznego to $O(n^2)$

W pesymistycznym przypadku element osiowy za każdym razem dzieli tablicę na jedną dużą i jedną pustą podtablicę. Wielkość dużej podtablicy jest o 1 mniejsza od wersji sprzed podziału. Złożoność algorytmu wynosi więc $(n - 1) + (n - 2) + \ldots + 2 + 1 = O(n^2)$.

W przypadku optymistycznym element osiowy za każdym razem dzieli tablicę na mniej więcej równe części. Niech $T(n)$ oznacza czas potrzebny na posortowanie tablicy n elementów. Wtedy:



$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$$

Podobnie jak w analizach sortowania przez scalanie otrzymujemy $T(n) = O(n \log n)$.

złożoność dla przypadku średniego to O(nlogn)

Średnio element osiowy nie dzieli za każdym razem tablicy na podtablice tej samej wielkości lub na pustą i dużą podtablicę. Jednak statystycznie wielkości obu części są zbliżone. Dlatego średnio czas wynosi $O(n \log n)$. Szczegółowe analizy przypadku średniego wykraczają poza zakres tej książki.

sortowanie szybkie a sortowanie przez scalanie

Sortowanie przez scalanie i sortowanie szybkie są oparte na podejściu dziel i rządź. W sortowaniu przez scalanie większość pracy to scalanie dwóch podlist wykonywane *po* ich posortowaniu. W sortowaniu szybkim najbardziej wymagający jest podział listy na podlisty, co dzieje się *przed* sortowaniem podlist. W przypadku pesymistycznym sortowanie przez scalanie jest wydajniejsze od sortowania szybkiego, ale w przypadku średnim złożoność obu technik jest taka sama. Scalanie przez sortowanie wymaga tablicy tymczasowej do sortowania podtablic. W sortowaniu szybkim nie potrzeba pamięci na dodatkową tablicę. Dlatego sortowanie szybkie jest wydajniejsze ze względu na pamięć.

23.5.1. Opisz sortowanie szybkie. Jaka jest złożoność czasowa tej techniki?

23.5.2. Dlaczego sortowanie szybkie jest wydajniejsze pamięciowo od sortowania przez scalanie?

23.5.3. Wzorując się na rysunku 23.7, pokaż przebieg sortowania szybkiego listy {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.5.4. Czy kod będzie działał, jeśli usuniesz wiersze 37. i 38. z programu do sortowania szybkiego? Podaj kontrprzykład pokazujący, że kod nie zadziała.

23.6. Sortowanie przez kopcowanie

W sortowaniu przez kopcowanie używany jest binarny kopiec. Najpierw wszystkie elementy są dodawane do kopca, a następnie usuwane są kolejno największe elementy, aby uzyskać posortowaną listę.

sortowanie przez kopcowanie
korzeń
lewe poddrzewo
prawe poddrzewo
długość
głębokość
liść

W *sortowaniu przez kopcowanie* używany jest binarny kopiec, czyli kompletne drzewo binarne. Drzewo binarne jest strukturą hierarchiczną. Jest albo puste, albo zawiera element nazywany *korzeniem* i dwa różne drzewa binarne — *lewe poddrzewo* i *prawe poddrzewo*. *Długość* ścieżki to liczba krawędzi w drzewie. *Głębokość* węzła to długość ścieżki z korzenia do tego węzła. Węzeł, który nie ma poddrzew, to *liść*.

Kopiec binarny jest drzewem binarnym o następujących cechach:

- kształt — kompletne drzewo binarne,
- kopiec — każdy węzeł jest większy lub równy względem wszystkich jego podwęzłów.

kompletne drzewo binarne

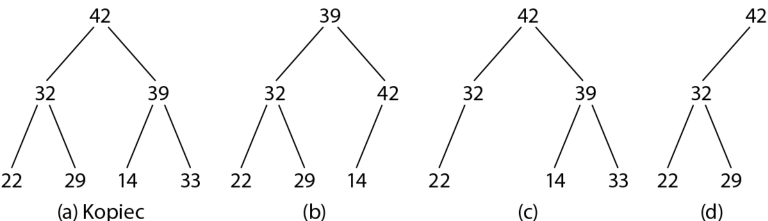
Drzewo binarne jest *kompletne*, jeśli jest na każdym poziomie pełne (wyjątkiem jest ostatni poziom) i wszystkie liście na ostatnim poziomie są umieszczone od lewej strony. Na rysunku 23.9 drzewa binarne (a) i (b) są kompletne, ale drzewa binarne (c) i (d) już nie. Ponadto drzewo binarne (a) jest kopcem, ale drzewo binarne (b) już nie, ponieważ korzeń (39) jest mniejszy niż prawe dziecko (42).

kopiec

Uwaga

Kopiec to pojęcie mające w informatyce wiele znaczeń. W tym rozdziale oznacza kopiec binarny.

Kup książkę Poleć książkę

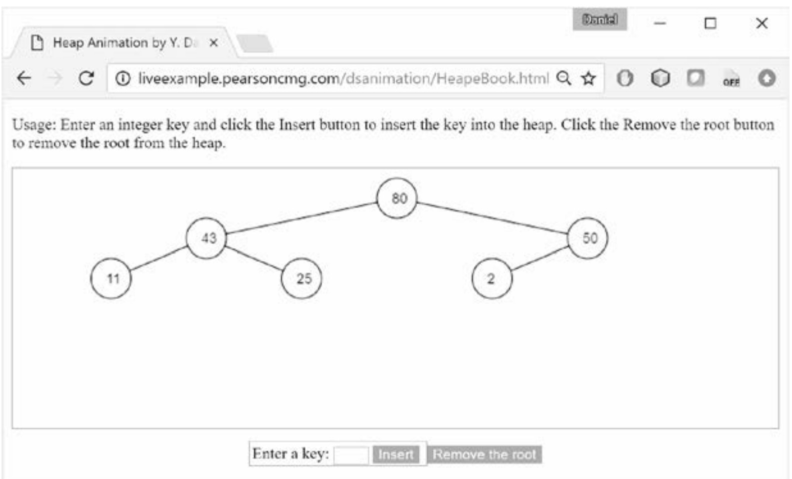


RYSUNEK 23.9. Kopiec binarny jest specjalną odmianą kompletnego drzewa binarnego

animacja działania kopca w witrynie

Uwaga edukacyjna

Kopiec można wydajnie zaimplementować na potrzeby wstawiania kluczy i usuwania korzenia. Interaktywną ilustrację działania kopca znajdziesz na stronie *http://liveexample.pearsoncmg.com/dsanimation/HeapeBook.html* (rysunek 23.10).

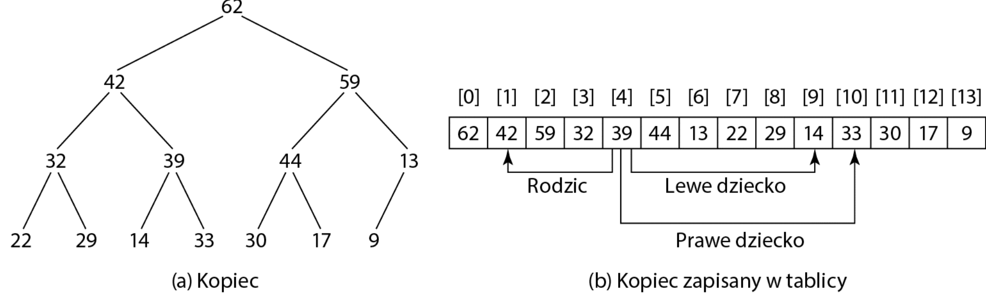


RYSUNEK 23.10. Narzędzie do animacji działania kopca umożliwia wstawianie kluczy i usuwanie korzenia

23.6.1. Przechowywanie kopca

Kopiec można przechowywać na liście `ArrayList` lub w tablicy, jeśli jego wielkość jest z góry znana. Kopiec z rysunku 23.11a można zapisać w tablicy z rysunku 23.11b. Korzeń znajduje się na pozycji 0, a dwójka jego dzieci — na pozycjach 1 i 2. Dla węzła z pozycji i jego lewe dziecko znajduje się na pozycji $2i + 1$, prawe dziecko na pozycji $2i + 2$, a rodzic na pozycji $(i - 1)/2$. Węzeł z elementem 39 znajduje się na pozycji 4, dlatego jego lewe dziecko (element 14) jest zapisany na pozycji 9 ($2 \times 4 + 1$), prawe dziecko (element 33) na pozycji 10 ($2 \times 4 + 2$), a rodzic (element 42) na pozycji 1 ($(4 - 1)/2$).

Kup ksi k   Pole ksi k



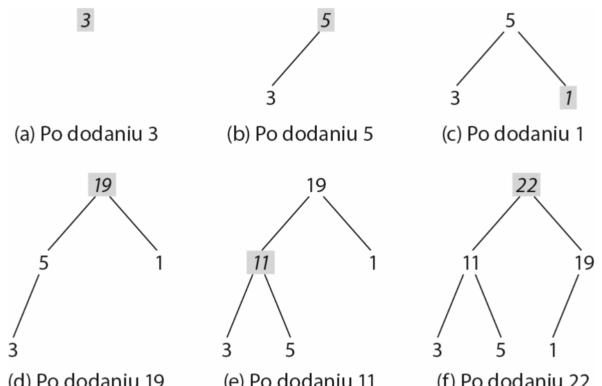
RYSUNEK 23.11. Kopiec binarny można zaimplementować za pomocą tablicy

23.6.2. Dodawanie węzła

Aby dodać do kopca nowy węzeł, najpierw umieść go na końcu kopca, a następnie zmodyfikuj drzewo:

```
Użyj ostatniego węzła jako bieżącego;
while (bieżący węzeł jest większy niż jego rodzic) {
  Przestaw bieżący węzeł z rodzicem;
  Bieżący węzeł znajduje się o poziom wyżej;
}
```

Przyjmij, że początkowo kopiec jest pusty. Na rysunku 23.12 widać kopiec po dodaniu liczb 3, 5, 1, 19, 11 i 22 w tej kolejności.



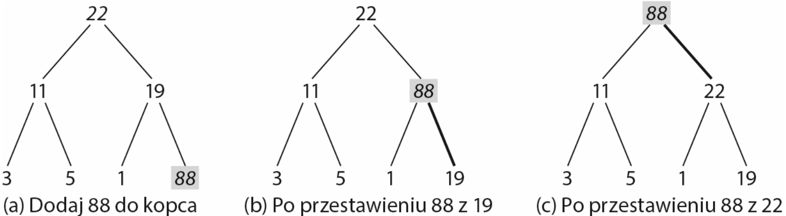
RYSUNEK 23.12. Elementy 3, 5, 1, 19, 11 i 22 są wstawiane do kopca

Teraz rozważ dodawanie 88 do kopca. Należy umieścić tę wartość na końcu drzewa (rysunek 23.13c), przestawić 88 z 19 (rysunek 23.13b) i przestawić 88 z 22 (rysunek 23.13c).

23.6.3. Usuwanie korzenia

Często trzeba usunąć znajdujący się w korzeniu maksymalny element. Po usunięciu korzenia drzewo trzeba zmodyfikować, by pozostało kopcem. Algorytm modyfikowania drzewa można opisać tak:

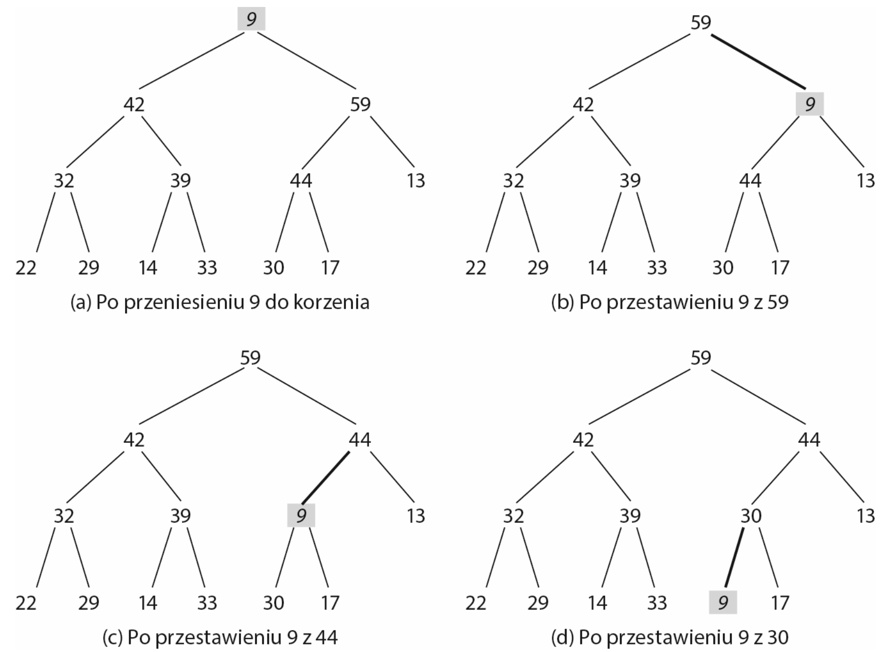
Kup ksi k Pole ksi k



RYSUNEK 23.13. Modyfikowanie kopca po dodaniu nowego węzła

```
Przenieś ostatni węzeł, aby zastąpić korzeń;
Użyj korzenia jako bieżącego węzła;
while (bieżący węzeł ma dzieci
       i jest mniejszy od któregoś z dzieci) {
  Przestaw bieżący węzeł z większym z dzieci;
  Bieżący węzeł znajduje się poziom niżej;
}
```

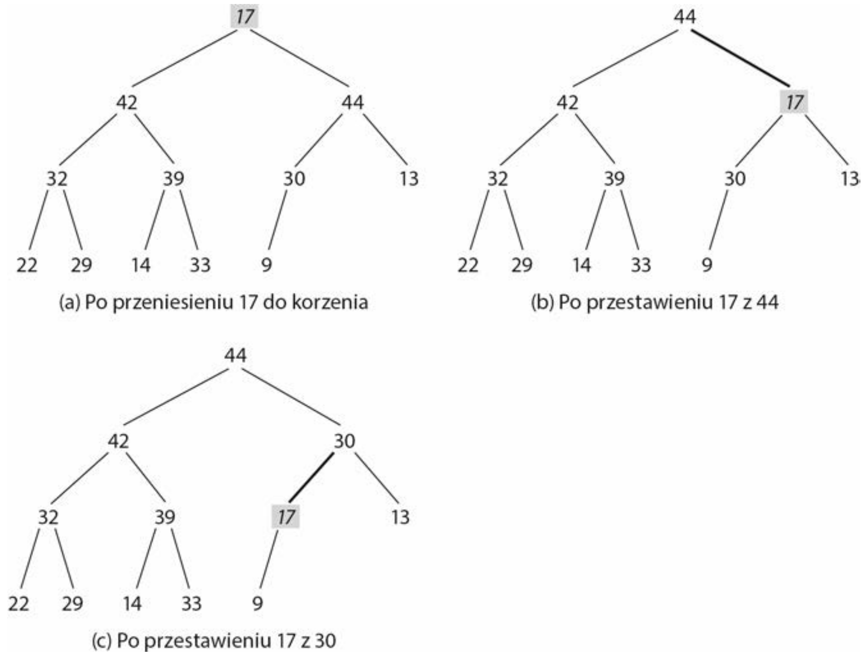
Na rysunku 23.14 pokazany jest proces modyfikowania kopca po usunięciu korzenia 62 z rysunku 23.11a. Jako korzeń ustawiany jest ostatni węzeł (9; rysunek 23.14a). Algorytm przestawia 9 z 59 (rysunek 23.14b), 9 z 44 (rysunek 23.14c) i 9 z 30 (rysunek 23.14d).



RYSUNEK 23.14. Modyfikowanie kopca po usunięciu korzenia 62

Rysunek 23.15 przedstawia modyfikowanie kopca po usunięciu korzenia 59 z rysunku 23.14d. Przenieś ostatni węzeł (17) do korzenia (rysunek 23.15a), przestaw 17 z 44 (rysunek 23.15b) i 17 z 30 (rysunek 23.15c).

Kup ksi k Pole ksi k



RYSUNEK 23.15. Modyfikowanie kopca po usunięciu korzenia 59

23.6.4. Klasa Heap

Teraz możesz zaprojektować i zaimplementować klasę `Heap`. Jej diagram jest pokazany na rysunku 23.16, a implementacja na listingu 23.9.

Heap<E>	
-list: java.util.ArrayList<E>	
-c: java.util.comparator<E>	
+Heap()	Tworzy domyślny pusty kopiec
+Heap(c: java.util.Comparator<E>)	Tworzy pusty kopiec z podanym komparatorem
+Heap(objects: E[])	Tworzy kopiec z podanymi obiektami
+add(newObject: E): void	Dodaje nowy obiekt do kopca
+remove(): E	Usuwa i zwraca korzeń kopca
+getSize(): int	Zwraca wielkość kopca
+isEmpty(): boolean	Zwraca `true`, jeśli kopiec jest pusty

RYSUNEK 23.16. Klasa Heap udostępnia operacje do zarządzania kopcem

LISTING 23.9. Heap.java

wewnętrzna reprezentacja kopca
komparator

```
1 public class Heap<E> {
2   private java.util.ArrayList<E> list = new java.util.ArrayList<>();
3   private java.util.Comparator<? super E> c;
4
5   /** Tworzy domyślny kopiec */
```

Kup ksi k Pole ksi k

```
konstruktor bezargumentowy     6   public Heap() {
tworzenie komparatora          7     this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
                               8   }
                               9
                              10   /** Tworzy kopiec z podanym komparatorem */
konstruktor                   11   public Heap(java.util.Comparator<E> c) {
                              12     this.c = c;
                              13   }
                              14
                              15   /** Tworzy kopiec na podstawie tablicy obiektów */
konstruktor                   16   public Heap(E[] objects) {
                              17     this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
                              18     for (int i = 0; i < objects.length; i++)
                              19       add(objects[i]);
                              20   }
                              21
                              22   /** Dodawanie nowego obiektu do kopca */
dodawanie nowego obiektu      23   public void add(E newObject) {
dołączanie obiektu            24     list.add(newObject); // Dołączanie obiektu do kopca
                              25     int currentIndex = list.size() - 1; // Indeks ostatniego węzła
                              26
                              27     while (currentIndex > 0) {
                              28       int parentIndex = (currentIndex - 1) / 2;
                              29       // Przestawianie, jeśli bieżący obiekt jest większy niż rodzic
                              30       if (c.compare(list.get(currentIndex),
                              31           list.get(parentIndex)) > 0) {
przestawianie z rodzicem      32         E temp = list.get(currentIndex);
                              33         list.set(currentIndex, list.get(parentIndex));
                              34         list.set(parentIndex, temp);
                              35       }
                              36       else
teraz to kopiec               37         break; // Drzewo jest teraz kopcem
                              38
                              39       currentIndex = parentIndex;
                              40     }
                              41   }
                              42
                              43   /** Usuwanie korzenia z kopca */
usuwanie korzenia             44   public E remove() {
opróżnianie kopca             45     if (list.size() == 0) return null;
                              46
korzeń                        47     E removedObject = list.get(0);
nowy korzeń                   48     list.set(0, list.get(list.size() - 1));
usuwanie ostatniego           49     list.remove(list.size() - 1);
  elementu                    50
                              51     int currentIndex = 0;
dostosowanie drzewa           52     while (currentIndex < list.size()) {
                              53       int leftChildIndex = 2 * currentIndex + 1;
                              54       int rightChildIndex = 2 * currentIndex + 2;
                              55
                              56       // Znajdowanie maksymalnego dziecka
                              57       if (leftChildIndex >= list.size()) break; // Drzewo jest kopcem
                              58         int maxIndex = leftChildIndex;
                              59         if (rightChildIndex < list.size()) {
porównanie dwóch dzieci       60           if (c.compare(list.get(maxIndex),
```

Kup książkę Poleć książkę

przestawianie z większym dzieckiem

```
61              list.get(rightChildIndex)) < 0) {
62            maxIndex = rightChildIndex;
63          }
64        }
65
66        // Przestawianie, jeśli bieżący węzeł jest mniejszy od maksimum
67        if (c.compare(list.get(currentIndex),
68                      list.get(maxIndex)) < 0) {
69          E temp = list.get(maxIndex);
70          list.set(maxIndex, list.get(currentIndex));
71          list.set(currentIndex, temp);
72          currentIndex = maxIndex;
73        }
74        else
75          break; // Drzewo jest kopcem
76      }
77
78      return removedObject;
79    }
80
81    /** Pobieranie liczby węzłów w drzewie */
82    public int getSize() {
83      return list.size();
84    }
85
86    /** Zwraca true, jeśli kopiec jest pusty */
87    public boolean isEmpty() {
88      return list.size() == 0;
89    }
90 }
```

Kopiec jest reprezentowany wewnętrznie za pomocą listy (wiersz 2.). Można zastąpić klasę `ArrayList` inną strukturą danych, a kontrakt klasy `Heap` pozostanie bez zmian.

Elementy kopca można porównywać za pomocą komparatora lub (jeśli nie podasz komparatora) według porządku naturalnego.

Konstruktor bezargumentowy (wiersz 6.) tworzy pusty kopiec, a porównywanie odbywa się za pomocą komparatora opartego na porządku naturalnym. Ten komparator jest tworzony za pomocą wyrażenia lambda (wiersz 7.).

Metoda `add(E newObject)` (wiersze 23. – 41.) dołącza obiekt do drzewa, a następnie jeśli obiekt jest większy od jego rodzica, przestawia te elementy. Proces ten jest kontynuowany do czasu, gdy nowy obiekt stanie się korzeniem lub będzie nie większy niż rodzic.

Metoda `remove()` (wiersze 44. – 79.) usuwa i zwraca korzeń. Aby kopiec pozostał kopcem, metoda przenosi ostatni obiekt do korzenia i jeśli ów obiekt jest mniejszy od większego z dzieci, przestawia te elementy. Proces ten jest kontynuowany do czasu, gdy ostatni obiekt stanie się liściem lub będzie nie mniejszy od dzieci.

23.6.5. Sortowanie z użyciem klasy Heap

Aby posortować tablice za pomocą kopca, najpierw utwórz obiekt klasy `Heap`, dodaj wszystkie elementy do kopca za pomocą metody `add` i usuń wszystkie elementy z kopca za pomocą metody `remove`. Elementy należy usuwać malejąco. Na listingu 23.10 przedstawiony jest program do sortowania tablicy za pomocą kopca.

Kup ksi k Pole ksi k

LISTING 23.10. HeapSort.java

domyślna metoda heapSort

metoda heapSort z komparatorem

tworzenie kopca

dodawanie elementu

usuwanie elementu

wywołanie metody heapSort

```
 1 import java.util.Comparator;
 2
 3 public class HeapSort {
 4   /** Metoda heapSort */
 5   public static <E> void heapSort(E[] list) {
 6     // Tworzenie kopca liczb całkowitych
 7     heapSort(list, (e1, e2) -> ((Comparable<E>)e1).compareTo(e2));
 8   }
 9
10   /** Metoda heapSort */
11   public static <E> void heapSort(E[] list, Comparator<E> c) {
12     // Tworzenie kopca liczb całkowitych
13     Heap<E> heap = new Heap<>(c);
14
15     // Dodawanie elementów do kopca
16     for (int i = 0; i < list.length; i++)
17       heap.add(list[i]);
18
19     // Usuwanie elementów z kopca
20     for (int i = list.length - 1; i >= 0; i--)
21       list[i] = heap.remove();
22  }
23
24   /** Metoda testowa */
25   public static void main(String[] args) {
26     Integer[] list = {-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53};
27     heapSort(list);
28     for (int i = 0; i < list.length; i++)
29       System.out.print(list[i] + " ");
30   }
31 }
```

```
-44 -5 -4 -3 0 1 1 2 3 3 4 5 53
```

Dostępne są dwie wersje metody `heapSort`. Wersja `heapSort(E[] list)` (wiersz 5.) sortuje listę w porządku naturalnym z wykorzystaniem interfejsu `Comparable`. Metoda `heapSort(E[] list, Comparator<E> c)` (wiersz 11.) sortuje listę z użyciem podanego komparatora.

23.6.6. Złożoność czasowa sortowania przez kopcowanie

wysokość kopca

Skup się teraz na analizie złożoności czasowej sortowania przez kopcowanie. Niech h oznacza *wysokość* kopca o n elementach. Wysokość niepustego drzewa to długość ścieżki z korzenia do najdalszego węzła. *Wysokość* drzewa zawierającego 1 węzeł to 0. Zwyczajowo wysokość pustego drzewa jest podawana jako –1. Ponieważ kopiec jest kompletnym drzewem binarnym, na pierwszym poziomie znajduje się 1 węzeł (2^0), na drugim poziomie mamy 2 węzły (2^1), a na k-tym poziomie — 2^{k-1} węzłów. Poziom h ma więc 2^{h-1} węzłów, a ostatni poziom (h + 1) przynajmniej 1 węzeł, a maksymalnie 2^h węzłów. Tak więc:

$$1 + 2 + \ldots + 2^{h-1} < n \le 1 + 2 + \ldots + 2^{h-1} + 2^h$$

Kup ksi k Pole ksi k

sortowanie przez kopcowanie a sortowanie przez scalanie

Oznacza to, że:

$$2^{h-1} - 1 < n \leq 2^{h+1} - 1$$
$$2^{h} < n + 1 \leq 2^{h+1}$$
$$h < \log(n + 1) \leq h + 1$$

Mamy więc $h < \log(n + 1)$ i $h \geq \log(n + 1) - 1$. Z tego wynika, że $\log(n + 1) - 1 < h \leq \log(n + 1)$. Dlatego wysokość kopca to $O(\log n)$. Oto bardziej precyzyjne oszacowanie: można dowieść, że dla niepustego drzewa $h = \lfloor \log n \rfloor$.

złożoność w przypadku pesymistycznym to O(n logn)

Ponieważ metoda `add` używa ścieżki z liścia do korzenia, dodanie nowego elementu do kopca zajmuje najwyżej h kroków. Dlatego łączny czas tworzenia początkowego kopca dla tablicy n elementów wynosi $O(n \log n)$. Metoda `remove` używa ścieżki z korzenia do liścia, dlatego zmodyfikowanie kopca po usunięciu korzenia wymaga najwyżej h kroków. Metoda `remove` jest wywoływana n razy, dlatego łączny czas uzyskania posortowanej tablicy na podstawie kopca to $O(n \log n)$.

Sortowanie przez scalanie i przez kopcowanie zajmuje $O(n \log n)$ czasu. Sortowanie przez scalanie wymaga tablicy tymczasowej do scalania podtablic. Sortowanie przez kopcowanie nie wymaga pamięci na dodatkową tablicę. Dlatego sortowanie przez kopcowanie jest wydajniejsze pamięciowo niż sortowanie przez scalanie.

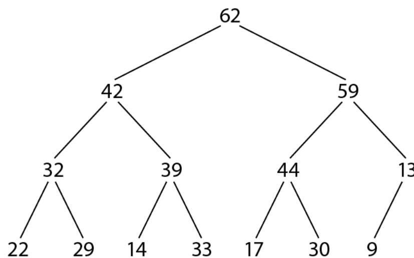
23.6.1. Czym jest kompletne drzewo binarne? Czym jest kopiec? Opisz usuwanie korzenia z kopca i dodawanie nowego obiektu do kopca.

23.6.2. Dodaj elementy 4, 5, 1, 2, 9 i 3 do kopca (w tej kolejności). Narysuj diagramy ilustrujące wygląd kopca po dodaniu każdego elementu.

23.6.3. Pokaż kroki tworzenia kopca z elementami {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.6.4. Przedstaw wygląd kopca po usunięciu korzenia z rysunku 23.15c.

23.6.5. Pokaż kroki usuwania wszystkich węzłów z poniższego kopca:



23.6.6. Które z poniższych instrukcji są błędne?

```
1 Heap<Object> heap1 = new Heap<>();
2 Heap<Number> heap2 = new Heap<>();
3 Heap<BigInteger> heap3 = new Heap<>();
4 Heap<Calendar> heap4 = new Heap<>();
5 Heap<String> heap5 = new Heap<>();
```

23.6.7. Jaką wartość zwraca metoda `remove`, gdy kopiec jest pusty?

23.6.8. Jaka jest złożoność czasowa dodawania nowego elementu do kopca? Jaka jest złożoność czasowa usuwania elementu z kopca?

23.6.9. Jaka jest wysokość niepustego kopca? Jaka jest wysokość kopców zawierających 16, 17 i 512 elementów? Jeśli kopiec ma wysokość 5, ile maksymalnie węzłów może zawierać?

Kup ksi k Pole ksi k

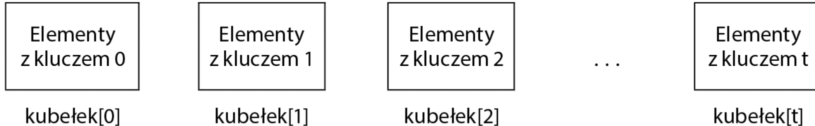
23.7. Sortowanie kubełkowe i pozycyjne

Sortowanie kubełkowe i pozycyjne to wydajne algorytmy sortowania liczb całkowitych.

Wszystkie algorytmy sortowania opisane do tej pory są algorytmami ogólnymi, działającymi dla kluczy dowolnego typu (liczb całkowitych, łańcuchów znaków i dowolnych obiektów umożliwiających porównywanie). Te algorytmy sortują elementy, porównując klucze. Udowodniono, że żaden algorytm oparty na porównaniach nie ma wydajności wyższej niż $O(n \log n)$. Jeśli jednak kluczami są liczby całkowite, można zastosować sortowanie kubełkowe bez konieczności porównywania kluczy.

sortowanie kubełkowe

Algorytm *sortowania kubełkowego* działa tak: przyjmij, że klucze należą do przedziału od 0 do t. Potrzebnych jest $t + 1$ kubełków o oznaczeniach 0, 1, …, t. Jeśli klucz elementu to i, element trafia do kubełka i. Każdy kubełek przechowuje elementy z kluczami o tej samej wartości.



Jako kubełków można użyć listy `ArrayList`. Algorytm sortowania kubełkowego listy elementów można opisać tak:

```
public static void bucketSort(E[] list) {
  E[] bucket = (E[])new java.util.ArrayList[t+1];

  // Rozdzielanie elementów listy do kubełków
  for (int i = 0; i < list.length; i++) {
    int key = list[i].getKey(); // Przyjmij, że element udostępnia metodę getKey()

    if (bucket[key] == null)
      bucket[key] = new java.util.ArrayList<>();

    bucket[key].add(list[i]);
  }

  // Przenoszenie elementów z kubełków z powrotem na listę
  int k = 0; // k to indeks listy
  for (int i = 0; i < bucket.length; i++) {
    if (bucket[i] != null) {
      for (int j = 0; j < bucket[i].size(); j++)
        list[k++] = bucket[i].get(j);
    }
  }
}
```

Posortowanie listy wymaga $O(n + t)$ czasu i $O(n + t)$ pamięci (n to wielkość listy).

Zauważ, że dla zbyt dużego t sortowanie kubełkowe jest niewskazane. Lepiej jest użyć wtedy sortowania pozycyjnego, które jest oparte na sortowaniu kubełkowym, ale używa tylko 10 kubełków.

stabilność

Warto zauważyć, że sortowanie kubełkowe jest *stabilne*. To oznacza, że jeśli dwa elementy pierwotnej listy mają ten sam klucz, ich kolejność na posortowanej liście się nie zmienia. Jeśli elementy e_1 i e_2 mają ten sam klucz i na pierwotnej liście e_1 znajduje się przed e_2, na posortowanej liście e_1 też znajdzie się przed e_2.

Kup ksi k Pole ksi k

sortowanie pozycyjne

Przyjmij, że kluczami są dodatnie liczby całkowite. *Sortowanie pozycyjne* polega na podziale kluczy na podgrupy za pomocą kubełków na podstawie cyfr z określonych pozycji. Należy wielokrotnie stosować sortowanie kubełkowe kluczy na podstawie cyfr z określonych pozycji, począwszy od pozycji najmniej istotnej.

animacja sortowania pozycyjnego w witrynie

Rozważ sortowanie pozycyjne elementów o następujących kluczach:

```
331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9
```

Należy wykonać sortowanie kubełkowe według ostatniej pozycji. Elementy są umieszczane w kubełkach:

kubełek[0]	kubełek[1]	kubełek[2]	kubełek[3]	kubełek[4]	kubełek[5]	kubełek[6]	kubełek[7]	kubełek[8]	kubełek[9]
230	331 231		343 453	454 34	45 345				59 9

Kolejność elementów pobranych z kubełków jest taka:

```
230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9
```

Teraz sortowanie kubełkowe jest stosowane według przedostatniej pozycji i elementy trafiają do kubełków:

kubełek[0]	kubełek[1]	kubełek[2]	kubełek[3]	kubełek[4]	kubełek[5]	kubełek[6]	kubełek[7]	kubełek[8]	kubełek[9]
9			230 331 231 34	343 45 345	453 454 59				

Kolejność elementów pobranych z kubełków jest taka:

```
9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59
```

Zauważ, że 9 jest traktowane jak 009.

Następnie sortowanie kubełkowe jest stosowane do pozycji trzeciej od końca. Elementy trafiają do kubełków:

kubełek[0]	kubełek[1]	kubełek[2]	kubełek[3]	kubełek[4]	kubełek[5]	kubełek[6]	kubełek[7]	kubełek[8]	kubełek[9]
9 34 45 59		230 231	331 343 345	453 454					

Oto kolejność elementów pobranych z kubełków:

```
9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454
```

Elementy są teraz posortowane.

Sortowanie pozycyjne n elementów z kluczami całkowitoliczbowymi wymaga $O(dn)$ czasu, gdzie d to maksymalna liczba pozycji w kluczach.

23.7.1. Czy za pomocą sortowania kubełkowego można posortować listę łańcuchów znaków?

23.7.2. Przedstaw przebieg sortowania pozycyjnego liczb 454, 34, 23, 43, 74, 86 i 76.

23.8. Sortowanie zewnętrzne

Sortowanie zewnętrzne umożliwia sortowanie dużych ilości danych.

We wszystkich algorytmach sortowania omawianych w poprzednich podrozdziałach obowiązywało założenie, że całe sortowane dane mieszczą się w pamięci wewnętrznej, na przykład w tablicy. Aby posortować dane zapisane w zewnętrznym pliku, trzeba najpierw przenieść dane do pamięci i posortować je wewnętrznie. Jeśli jednak plik jest zbyt duży, nie da się jednocześnie umieścić wszystkich danych w pamięci. W tym podrozdziale opisane jest sortowanie danych z dużych zewnętrznych plików. Służy do tego *sortowanie zewnętrzne*.

sortowanie zewnętrzne

Dla uproszczenia przyjmij, że w pliku binarnym *largedata.dat* zapisane są 2 miliony wartości typu `int`. Ten plik jest tworzony za pomocą programu z listingu 23.11.

LISTING 23.11. CreateLargeFile.java

```
1 import java.io.*;
2
3 public class CreateLargeFile {
4   public static void main(String[] args) throws Exception {
5     DataOutputStream output = new DataOutputStream(
6       new BufferedOutputStream(
7       new FileOutputStream("largedata.dat"));
8
9     for (int i = 0; i < 2_000_000; i++)
10      output.writeInt((int)(Math.random() * 1000000));
11
12    output.close();
13
14    // Wyświetlanie pierwszych 100 liczb
15    DataInputStream input = new DataInputStream(
16      new BufferedInputStream(new FileInputStream("largedata.dat")));
17    for (int i = 0; i < 100; i++)
18      System.out.print(input.readInt() + " ");
19
20    input.close();
21  }
22 }
```

binarny strumień wyjściowy (wiersz 5); zapis wartości typu int (wiersz 10); zamykanie pliku wyjściowego (wiersz 12); odczyt wartości typu int (wiersz 18); zamykanie pliku wejściowego (wiersz 20)

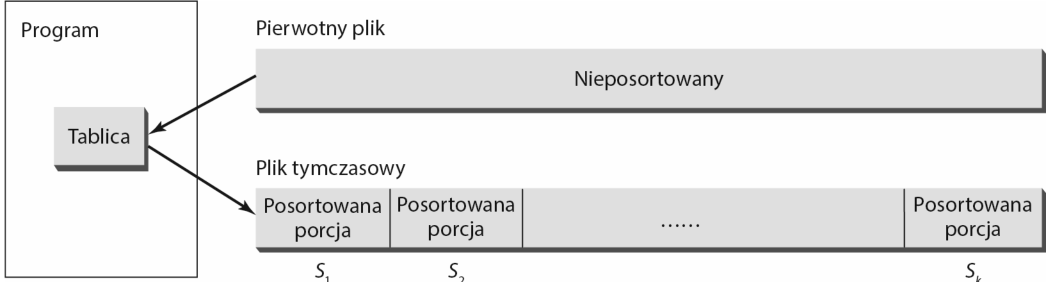
```
569193 131317 608695 776266 767910 624915 458599 5010 ... itd.
```

Do posortowania tego pliku w dwóch etapach można użyć odmiany sortowania przez scalanie:

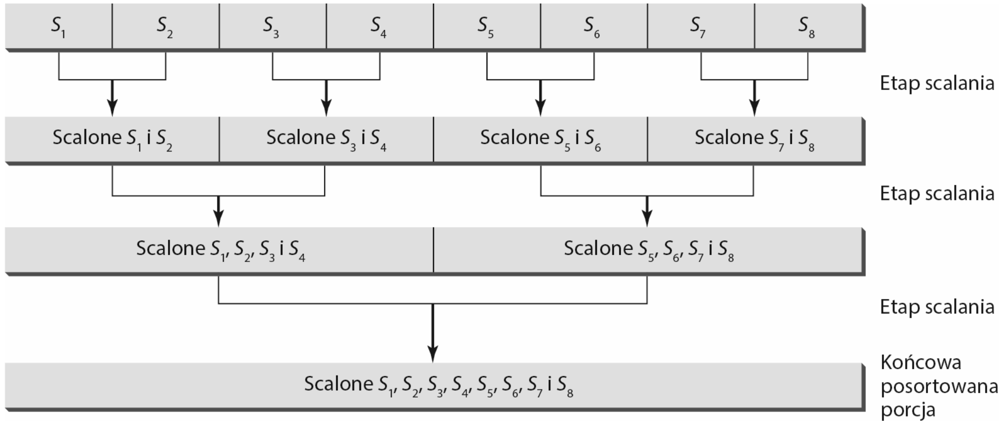
Etap 1. Wielokrotne wczytywanie danych z pliku do tablicy, sortowanie tablicy za pomocą wewnętrznego algorytmu sortowania i zapisywanie danych z tablicy w pliku tymczasowym. Ten proces jest pokazany na rysunku 23.17. Najlepiej jest utworzyć dużą tablicę, ale jej maksymalna wielkość zależy od ilości pamięci przydzielonej maszynie JVM przez system operacyjny. Przyjmij, że maksymalna wielkość tablicy to 100 000 wartości typu `int`. W pliku tymczasowym każde 100 000 takich wartości jest posortowanych. Te porcje są oznaczone jako S_1, S_2, ..., S_k, przy czym ostatnia porcja, S_k, może zawierać mniej niż 100 000 wartości.

Etap 2. Scalane są pary posortowanych porcji (S_1 z S_2, S_3 z S_4 itd.), aby uzyskać większą posortowaną porcję zapisywaną w nowym pliku tymczasowym. Proces ten jest kontynuowany do czasu otrzymania jednej dużej posortowanej porcji. Na rysunku 23.18 pokazano scalanie ośmiu porcji.

Kup książkę Poleć książkę



RYSUNEK 23.17. Pierwotny plik jest sortowany porcjami



RYSUNEK 23.18. Posortowane porcje są iteracyjnie scalane

Uwaga

Nie trzeba scalać dwóch kolejnych porcji. Możesz na przykład scalić S_1 z S_5, S_2 z S_6, S_3 z S_7 i S_4 z S_8. Ta obserwacja przyda się do wydajnego zaimplementowania etapu 2.

23.8.1. Implementowanie etapu 1.

Na listingu 23.12 pokazana jest metoda, która wczytuje każdą porcję danych z pliku, sortuje porcje i zapisuje posortowane porcje w nowym pliku. Metoda zwraca liczbę porcji.

LISTING 23.12. Tworzenie posortowanych porcji

pierwotny plik

plik z posortowanymi porcjami

```
 1 /** Sortowanie pierwotnego pliku w posortowane porcje */
 2 private static int initializeSegments
 3     (int segmentSize, String originalFile, String f1)
 4      throws Exception {
 5   int[] list = new int[segmentSize];
 6   DataInputStream input = new DataInputStream(
 7     new BufferedInputStream(new FileInputStream(originalFile)));
 8   DataOutputStream output = new DataOutputStream(
 9     new BufferedOutputStream(new FileOutputStream(f1)));
10
```

Kup ksi k   Pole ksi k

```
11   int numberOfSegments = 0;
12   while (input.available() > 0) {
13     numberOfSegments++;
14     int i = 0;
15     for ( ; input.available() > 0 && i < segmentSize; i++) {
16       list[i] = input.readInt();
17     }
18
19     // Sortowanie tablicy list[0..i–1]
20     java.util.Arrays.sort(list, 0, i);                      // sortowanie porcji
21
22     // Zapis tablicy w pliku f1.dat
23     for (int j = 0; j < i; j++) {
24       output.writeInt(list[j]);                             // zapis w pliku
25     }
26   }
27
28   input.close();                                            // zamykanie pliku
29   output.close();
30
31   return numberOfSegments;                                  // zwraca liczbę porcji
32 }
```

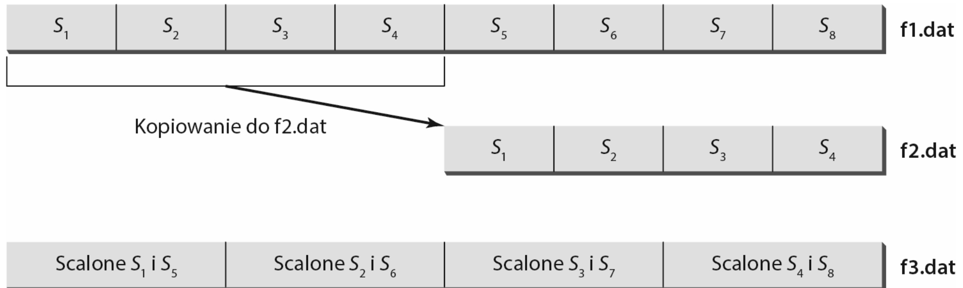
Ta metoda tworzy tablicę o maksymalnej wielkości (wiersz 5.), strumień wejściowy powiązany z pierwotnym plikiem (wiersz 6.) i strumień wyjściowy powiązany z plikiem tymczasowym (wiersz 8.). W celu poprawy wydajności używane są strumienie buforowane.

Kod z wierszy 14. – 17. wczytuje porcję danych z pliku do tablicy. W wierszu 20. tablica jest sortowana. Wiersze 23. – 25. zapisują dane z tablicy do pliku tymczasowego.

W wierszu 31. zwracana jest liczba porcji. Zauważ, że każda porcja oprócz ostatniej ma `MAX_ARRAY_SIZE` elementów (w ostatniej porcji elementów może być mniej).

23.8.2. Implementowanie etapu 2.

W każdym kroku scalania dwie posortowane porcje są scalane w nową porcję. Po każdym kroku scalania wielkość nowych porcji jest podwajana, a liczba porcji spada o połowę. Porcje stają się teraz za duże, aby je pomieścić w pamięci. Na etapie scalania należy skopiować połowę porcji z pliku *f1.dat* do pliku tymczasowego *f2.dat*, a następnie scalić pierwszą pozostałą porcję z *f1.dat* z pierwszą porcją z *f2.dat* i zapisać wynik w pliku tymczasowym *f3.dat*, co ilustruje rysunek 23.19.



RYSUNEK 23.19. Posortowane porcje są iteracyjnie scalane

Uwaga

W *f1.dat* może znajdować się jedna porcja więcej niż w *f2.dat*. Wtedy po scalaniu przenieś ostatnią porcję do *f3.dat*.

Metoda z listingu 23.13 kopiuje pierwszą połowę porcji z *f1.dat* do *f2.dat*. Metoda z listingu 23.14 scala wszystkie pary porcji z *f1.dat* i *f2.dat*. Metoda z listingu 23.15 scala dwie porcje.

LISTING 23.13. Kopiowanie pierwszej połowy porcji

strumień wejściowy f1 (wiersz 2)
kopiowanie porcji (wiersz 5)

```
1 private static void copyHalfToF2(int numberOfSegments,
2     int segmentSize, DataInputStream f1, DataOutputStream f2)
3     throws Exception {
4   for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
5     f2.writeInt(f1.readInt());
6   }
7 }
```

LISTING 23.14. Scalanie wszystkich porcji

strumienie wejściowe f1 i f2 (wiersz 2)
strumień wyjściowy f3 (wiersz 3)
scalanie 2 porcji (wiersz 5)
dodatkowa porcja w f1? (wiersz 9)

```
 1 private static void mergeSegments(int numberOfSegments,
 2     int segmentSize, DataInputStream f1, DataInputStream f2,
 3     DataOutputStream f3) throwsException {
 4   for (int i = 0; i < numberOfSegments; i++) {
 5     mergeTwoSegments(segmentSize, f1, f2, f3);
 6   }
 7
 8   // Jeśli f1 zawiera dodatkową porcję, należy skopiować ją do f3
 9   while (f1.available() > 0) {
10     f3.writeInt(f1.readInt());
11   }
12 }
```

LISTING 23.15. Scalanie dwóch porcji

strumienie wejściowe f1 i f2 (wiersz 2)
strumień wyjściowy f3 (wiersz 3)
odczyt z f1 (wiersz 4)
odczyt z f2 (wiersz 5)
zapis w f3 (wiersz 11)
koniec porcji z f1 (wiersz 14)
zapis w f3 (wiersz 21)

```
 1 private static void mergeTwoSegments(int segmentSize,
 2     DataInputStream f1, DataInputStream f2,
 3     DataOutputStream f3) throws Exception {
 4   int intFromF1 = f1.readInt();
 5   int intFromF2 = f2.readInt();
 6   int f1Count = 1;
 7   int f2Count = 1;
 8
 9   while (true) {
10     if (intFromF1 < intFromF2) {
11       f3.writeInt(intFromF1);
12       if (f1.available() == 0|| f1Count++ >= segmentSize) {
13         f3.writeInt(intFromF2);
14         break;
15       }
16       else {
17         intFromF1 = f1.readInt();
18       }
19     }
20     else {
21       f3.writeInt(intFromF2);
```

Kup ksi k Pole ksi k

koniec porcji z f2

reszta porcji z f1

reszta porcji z f2

```
22        if (f2.available() == 0 || f2Count++ >= segmentSize) {
23          f3.writeInt(intFromF1);
24          break;
25        }
26        else {
27          intFromF2 = f2.readInt();
28        }
29      }
30    }
31
32    while (f1.available() > 0 && f1Count++ < segmentSize) {
33      f3.writeInt(f1.readInt());
34    }
35
36    while (f2.available() > 0 && f2Count++ < segmentSize) {
37      f3.writeInt(f2.readInt());
38    }
39 }
```

23.8.3. Łączenie dwóch etapów

Na listingu 23.16 pokazany jest kompletny program do sortowania wartości typu `int` z pliku *largedata.dat* i zapisywania posortowanych danych w pliku *sortedfile.dat*.

LISTING 23.16. SortLargeFile.java

maksymalna wielkość tablicy

wielkość bufora strumienia I/O

tworzenie początkowych porcji

rekurencyjne scalanie

```
 1 import java.io.*;
 2
 3 public class SortLargeFile {
 4   public static final int MAX_ARRAY_SIZE = 100000;
 5   public static final int BUFFER_SIZE = 100000;
 6
 7   public static void main(String[] args) throws Exception {
 8     // Sortowanie danych z largedata.dat i zapisywanie ich w pliku sortedfile.dat
 9     sort("largedata.dat", "sortedfile.dat");
10
11     // Wyświetlanie pierwszych 100 liczb z posortowanego pliku
12     displayFile("sortedfile.dat");
13   }
14
15   /** Sortowanie danych z pliku źródłowego i ich zapis w pliku docelowym */
16   public static void sort(String sourcefile, String targetfile)
17       throws Exception {
18     // Implementacja etapu 1. Tworzenie porcji
19     int numberOfSegments =
20       initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");
21
22     // Implementacja etapu 2. Rekurencyjne scalanie porcji
23     merge(numberOfSegments, MAX_ARRAY_SIZE,
24       "f1.dat", "f2.dat", "f3.dat", targetfile);
25   }
26
27   /** Dzieli pierwotny plik na posortowane porcje */
28   private static int initializeSegments
29       (int segmentSize, String originalFile, String f1)
```

Kup książkę Poleć książkę

```
30        throws Exception {
31      // Pominięte (to samo co na listingu 23.12)
32    }
33
34    private static void merge(int numberOfSegments, int segmentSize,
35        String f1, String f2, String f3, String targetfile)
36        throws Exception {
37      if (numberOfSegments > 1) {
38        mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
39        merge((numberOfSegments + 1) / 2, segmentSize * 2,
40          f3, f1, f2, targetfile);
41      }
42      else { // Zmiana nazwy f1 na sortedfile.dat
43        File sortedFile = new File(targetfile);
44        if (sortedFile.exists()) sortedFile.delete();
45        new File(f1).renameTo(sortedFile);
46      }
47    }
48
49    private static void mergeOneStep(int numberOfSegments,
50        int segmentSize, String f1, String f2, String f3)
51        throws Exception {
52      DataInputStream f1Input = new DataInputStream(
53        new BufferedInputStream(new FileInputStream(f1), BUFFER_SIZE));
54      DataOutputStream f2Output = new DataOutputStream(
55        new BufferedOutputStream(new FileOutputStream(f2), BUFFER_SIZE));
56
57      // Kopiowanie połowy porcji z f1.dat do f2.dat
58      copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
59      f2Output.close();
60
61      // Scalanie pozostałych porcji z f1 z porcjami z f2 w f3
62      DataInputStream f2Input = new DataInputStream(
63        new BufferedInputStream(new FileInputStream(f2), BUFFER_SIZE));
64      DataOutputStream f3Output = new DataOutputStream(
65        new BufferedOutputStream(new FileOutputStream(f3), BUFFER_SIZE));
66
67      mergeSegments(numberOfSegments / 2,
68        segmentSize, f1Input, f2Input, f3Output);
69
70      f1Input.close();
71      f2Input.close();
72      f3Output.close();
73    }
74
75    /** Kopiowanie pierwszej połowy porcji z f1.dat do f2.dat */
76    private static void copyHalfToF2(int numberOfSegments,
77        int segmentSize, DataInputStream f1, DataOutputStream f2)
78        throws Exception {
79      // Pominięte (to samo co na listingu 23.13)
80    }
81
82    /** Scalanie wszystkich porcji */
83    private static void mergeSegments(int numberOfSegments,
84        int segmentSize, DataInputStream f1, DataInputStream f2,
```

- 38: jeden etap scalania
- 39: rekurencyjne scalanie
- 43: końcowy posortowany plik
- 52: strumień wejściowy f1Input
- 54: strumień wyjściowy f2Output
- 58: kopiowanie połowy porcji do f2
- 59: zamykanie f2Output
- 62: strumień wejściowy f2Input
- 64: strumień wyjściowy f3Output
- 67: scalanie dwóch porcji
- 70: zamykanie strumieni

Kup ksi k Pole ksi k

wyświetlanie pliku

```
85        DataOutputStream f3) throws Exception {
86      // Pominięte (to samo co na listingu 23.14)
87    }
88
89    /** Scalanie dwóch porcji */
90    private static void mergeTwoSegments(int segmentSize,
91        DataInputStream f1, DataInputStream f2,
92        DataOutputStream f3) throws Exception {
93      // Pominięte (to samo co na listingu 23.15)
94    }
95
96    /** Wyświetlanie pierwszych 100 liczb z podanego pliku */
97    public static void displayFile(String filename) {
98      try {
99        DataInputStream input =
100         new DataInputStream(new FileInputStream(filename));
101       for (int i = 0; i < 100; i++)
102         System.out.print(input.readInt() + " ");
103       input.close();
104     }
105     catch (IOException ex) {
106       ex.printStackTrace();
107     }
108   }
109 }
```

```
0 1 1 1 2 2 2 3 3 4 5 6 8 8 9 9 9 10 10 11 itd.
```

Przed uruchomieniem tego programu wykonaj kod z listingu 23.11, *CreateLargeFile.java*, aby utworzyć plik *largedata.dat*. Wywołanie `sort("largedata.dat", "sortedfile.dat")` (wiersz 9.) wczytuje dane z pliku *largedata.dat* i zapisuje posortowane dane w pliku *sortedfile.dat*. Wywołanie `displayFile("sortedfile.dat")` (wiersz 12.) wyświetla pierwszych 100 liczb z podanego pliku. Zauważ, że pliki są tworzone za pomocą binarnych operacji I/O. Nie można ich wyświetlić za pomocą edytora tekstu takiego jak Notatnik.

Metoda `sort` najpierw tworzy początkowe porcje na podstawie pierwotnej tablicy i zapisuje posortowane porcje w nowym pliku, *f1.dat* (wiersze 19. i 20.), a następnie tworzy posortowany plik `targetfile` (wiersze 23. i 24.).

Metoda `merge`:

```
merge(int numberOfSegments, int segmentSize,
  String f1, String f2, String f3, String targetfile)
```

scala porcje z `f1` w `f3`, używając pomocniczego pliku `f2`. Metoda `merge` jest wywoływana rekurencyjnie i działa w wielu krokach. Każdy krok powoduje zmniejszenie `numberOfSegments` o połowę i podwojenie wielkości posortowanych porcji. Po zakończeniu jednego kroku scalania następny krok powoduje scalenie nowych porcji z `f3` w `f2` z użyciem pliku pomocniczego `f1`. Nowe wywołanie metody `merge` wygląda tak:

```
merge((numberOfSegments + 1) / 2, segmentSize * 2,
  f3, f1, f2, targetfile);
```

Liczba porcji (`numberOfSegments`) w następnym kroku scalania wynosi `(numberOfSegments + 1) / 2`. Gdy `numberOfSegments` to 5, w następnym kroku scalania `numberOfSegments` jest równe 3, ponieważ każda para porcji jest scalana i pozostaje jedna niescalona porcja.

Kup ksi k Pole ksi k

Rekurencyjna metoda `merge` kończy pracę, gdy `numberOfSegments` to 1. Wtedy `f1` zawiera posortowane dane i nazwa pliku jest zmieniana na nazwę ze zmiennej `targetfile` (wiersz 45.).

23.8.4. Złożoność sortowania zewnętrznego

Przy sortowaniu zewnętrznym dominującym kosztem są operacje I/O. Przyjmij, że n to liczba sortowanych elementów z pliku. W etapie 1. n elementów jest wczytywanych z pierwotnego pliku i zapisywanych w pliku tymczasowym. Dlatego w etapie 1. złożoność operacji I/O wynosi $O(n)$.

W etapie 2., przed pierwszym krokiem scalania, liczba posortowanych porcji to $\frac{n}{c}$, gdzie c to `MAX_ARRAY_SIZE`. Każdy krok scalania zmniejsza liczbę porcji o połowę. Dlatego po pierwszym kroku scalania liczba porcji wynosi $\frac{n}{2c}$. Po drugim kroku scalania liczba porcji to $\frac{n}{2^2 c}$, a po trzecim — $\frac{n}{2^3 c}$. Po $\log(\frac{n}{c})$ krokach scalania liczba porcji jest redukowana do 1. Tak więc łączna liczba kroków scalania wynosi $\log(\frac{n}{c})$.

W każdym kroku scalania połowa porcji jest wczytywana z pliku `f1` i zapisywana w pliku tymczasowym `f2`. Porcje pozostawione w `f1` są scalane z porcjami z `f2`. Liczba operacji I/O w każdym kroku scalania wynosi $O(n)$. Ponieważ łączna liczba kroków scalania wynosi $\log(\frac{n}{c})$, łączna liczba operacji I/O to:

$$O(n) \times \log(\frac{n}{c}) = O(n \log n)$$

Tak więc złożoność sortowania zewnętrznego wynosi $O(n \log n)$.

23.8.1. Opisz działanie sortowania zewnętrznego. Jaka jest jego złożoność?

23.8.2. W pliku zewnętrznym *largedata.dat* przechowywanych jest 10 liczb: {2, 3, 4, 0, 5, 6, 7, 9, 8, 1}. Prześledź ręcznie działanie programu `SortLargeFile` przy `MAX_ARRAY_SIZE = 2`.

NAJWAŻNIEJSZE POJĘCIA

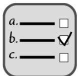
sortowanie bąbelkowe
sortowanie kubełkowe
kompletne drzewo binarne
sortowanie zewnętrzne
kopiec
sortowanie przez kopcowanie
wysokość kopca
sortowanie przez scalanie
sortowanie szybkie
sortowanie pozycyjne

PODSUMOWANIE ROZDZIAŁU

1. Złożoność pesymistyczna dla *sortowania przez wybieranie*, *sortowania przez wstawianie*, *sortowania bąbelkowego* i *sortowania szybkiego* wynosi $O(n^2)$.
2. Dla *sortowania przez scalanie* złożoność średnia i pesymistyczna wynoszą $O(n \log n)$. Średni czas działania sortowania szybkiego to $O(n \log n)$.
3. *Kopiec* to struktura danych przydatna do projektowania wydajnych algorytmów (na przykład sortowania). Nauczyłeś się, jak zdefiniować i zaimplementować klasę kopca oraz jak wstawiać i usuwać elementy w kopcu.
4. Złożoność czasowa *sortowania przez kopcowanie* to $O(n \log n)$.

Kup ksi k Pole ksi k

5. *Sortowanie kubełkowe* i *pozycyjne* to specjalne algorytmy do sortowania kluczy całkowitoliczbowych. Sortują one klucze za pomocą kubełków, a nie na podstawie porównań. Są wydajniejsze od ogólnych algorytmów sortowania.
6. *Sortowanie zewnętrzne* (jest to odmiana sortowania przez scalanie) można zastosować do sortowania dużych zbiorów danych z plików zewnętrznych.



Quiz

Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

Ćwiczenia programistyczne

Podrozdziały 23.3 – 23.5

23.1. *Generyczne sortowanie bąbelkowe.* Napisz dwie pokazane poniżej metody generyczne do sortowania bąbelkowego. Pierwsza ma sortować elementy z użyciem interfejsu `Comparable`, a druga — z użyciem interfejsu `Comparator`.

```
public static <E extends Comparable<E>>
  void bubbleSort(E[] list)
public static <E> void bubbleSort(E[] list,
  Comparator<? super E> comparator)
```

23.2. *Generyczne sortowanie przez scalanie.* Napisz dwie pokazane poniżej metody generyczne do sortowania przez scalanie. Pierwsza ma sortować elementy z użyciem interfejsu `Comparable`, a druga — z użyciem interfejsu `Comparator`.

```
public static <E extends Comparable<E>>
  void mergeSort(E[] list)
public static <E> void mergeSort(E[] list,
  Comparator<? super E> comparator)
```

23.3. *Generyczne sortowanie szybkie.* Napisz dwie pokazane poniżej metody generyczne do sortowania szybkiego. Pierwsza ma sortować elementy z użyciem interfejsu `Comparable`, a druga — z użyciem interfejsu `Comparator`.

```
public static <E extends Comparable<E>>
  void quickSort(E[] list)
public static <E> void quickSort(E[] list,
  Comparator<? super E> comparator)
```

23.4. *Ulepszone sortowanie szybkie.* Algorytm sortowania szybkiego przedstawiony w tej książce jako element osiowy wybiera pierwszy element z listy. Zmodyfikuj ten algorytm, aby używał środkowej wartości spośród pierwszego, środkowego i ostatniego elementu.

***23.5.** *Zmodyfikowane sortowanie przez scalanie.* Zmodyfikuj metodę `mergeSort`, aby rekurencyjnie sortowała pierwszą i drugą połowę tablicy bez tworzenia nowych tablic tymczasowych, a następnie scalała obie połowy w tablicy tymczasowej i kopiowała jej zawartość do pierwotnej tablicy (rysunek 23.6b).

23.6. *Sprawdzanie kolejności.* Napisz wymienione niżej przeciążone metody, aby sprawdzały, czy tablica jest uporządkowana rosnąco, czy malejąco. Domyślnie metoda wykrywa kolejność rosnącą. Aby wykrywać kolejność malejącą, przypisz wartość `false` do parametru `ascending`.

Kup ksi k   Pole ksi k

```
public static boolean ordered(int[] list)
public static boolean ordered(int[] list, boolean ascending)
public static boolean ordered(double[] list)
public static boolean ordered
  (double[] list, boolean ascending)
public static <E extends Comparable<E>>
  boolean ordered(E[] list)
public static <E extends Comparable<E>> boolean ordered
  (E[] list, boolean ascending)
public static <E> boolean ordered(E[] list,
  Comparator<? super E> comparator)
public static <E> boolean ordered(E[] list,
  Comparator<? super E> comparator, boolean ascending)
```

Podrozdział 23.6

kopiec maksymalny
kopiec minimalny

23.7. *Kopiec minimalny*. Kopiec opisany w tekście jest nazywany *maksymalnym*; każdy węzeł jest w nim większy lub równy względem dzieci. W *kopcu minimalnym* każdy węzeł jest mniejszy lub równy względem dzieci. Kopce minimalne często stosuje się do implementowania kolejek priorytetowych. Zmodyfikuj klasę `Heap` z listingu 23.9, aby zaimplementować kopiec minimalny.

23.8. *Generyczne sortowanie przez wstawianie*. Napisz dwie pokazane poniżej metody generyczne do sortowania przez wstawianie. Pierwsza ma sortować elementy z użyciem interfejsu `Comparable`, a druga — z użyciem interfejsu `Comparator`.

```
public static <E extends Comparable<E>>
  void insertionSort(E[] list)
public static <E> void insertionSort(E[] list,
  Comparator<? super E> comparator)
```

***23.9.** *Generyczne sortowanie przez kopcowanie*. Napisz dwie pokazane poniżej metody generyczne do sortowania przez kopcowanie. Pierwsza ma sortować elementy z użyciem interfejsu `Comparable`, a druga — z użyciem interfejsu `Comparator`. *Wskazówka*: użyj klasy `Heap` z ćwiczenia 23.5.

```
public static <E extends Comparable<E>>
  void heapSort(E[] list)
public static <E> void heapSort(E[] list,
  Comparator<? super E> comparator)
```

****23.10.** *Wizualizowanie kopca*. Napisz program, który wyświetla kopiec w formie graficznej (rysunek 23.10). Program ma umożliwiać wstawianie i usuwanie elementów kopca.

23.11. *Metody `clone` i `equals` w klasie `Heap`*. Zaimplementuj metody `clone` i `equals` w klasie `Heap`.

Użyj szablonu ze strony *http://liveexample.pearsoncmg.com/test/Exercise23_11.txt*.

Podrozdział 23.7

***23.12.** *Sortowanie pozycyjne*. Napisz program, który losowo generuje 1 000 000 liczb całkowitych i sortuje je za pomocą sortowania pozycyjnego.

***23.13.** *Czas sortowania*. Napisz program, który pobiera czas sortowania przez wybieranie, sortowania bąbelkowego, sortowania szybkiego, sortowania przez kopcowanie i sortowania pozycyjnego danych wejściowych liczących 50 000, 100 000, 150 000, 200 000, 250 000 i 300 000 elementów. Program powinien losowo generować dane i wyświetlać poniższą tabelę.

Kup ksi k   Pole ksi k

Wielkość tablicy	*Sortowanie przez wybieranie*	*Sortowanie bąbelkowe*	*Sortowanie przez scalanie*	*Sortowanie szybkie*	*Sortowanie przez kopcowanie*	*Sortowanie pozycyjne*
50 000						
100 000						
150 000						
200 000						
250 000						
300 000						

Wskazówka: do uzyskania czasu sortowania możesz użyć następującego szablonu:

```
long startTime = System.nanoTime();
Wykonywanie operacji;
long endTime = System.nanoTime();
long executionTime = endTime - startTime;
```

Podrozdział 23.8

***23.14.** *Czas sortowania zewnętrznego.* Napisz program, który oblicza czas sortowania zewnętrznego 5 000 000, 10 000 000, 15 000 000, 20 000 000, 25 000 000 i 30 000 000 liczb całkowitych. Program powinien wyświetlać tabelę:

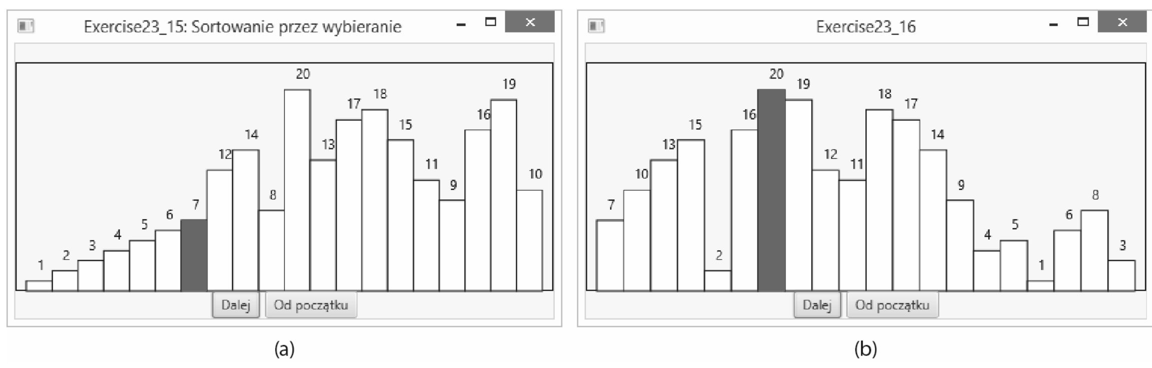
Wielkość pliku	*5 000 000*	*10 000 000*	*15 000 000*	*20 000 000*	*25 000 000*	*30 000 000*
Czas						

Ogólne

***23.15.** *Animacja sortowania przez wybieranie*. Napisz program wyświetlający animację sortowania przez wybieranie. Utwórz tablicę uporządkowanych losowo 20 różnych liczb od 1 do 20. Elementy tablicy należy wyświetlić w formie histogramu (rysunek 23.20a). Kliknięcie przycisku *Dalej* ma skutkować iteracją zewnętrznej pętli algorytmu i wyświetleniem histogramu ze zmodyfikowaną tablicą. Wyróżnij kolorem ostatni słupek posortowanej podtablicy. Po zakończeniu pracy algorytmu wyświetl informacje dla użytkownika. Przycisk *Od początku* ma tworzyć nową losową tablicę i umożliwić rozpoczęcie pracy od nowa. Możesz łatwo zmodyfikować ten program, aby wyświetlał animację sortowania przez wstawianie.

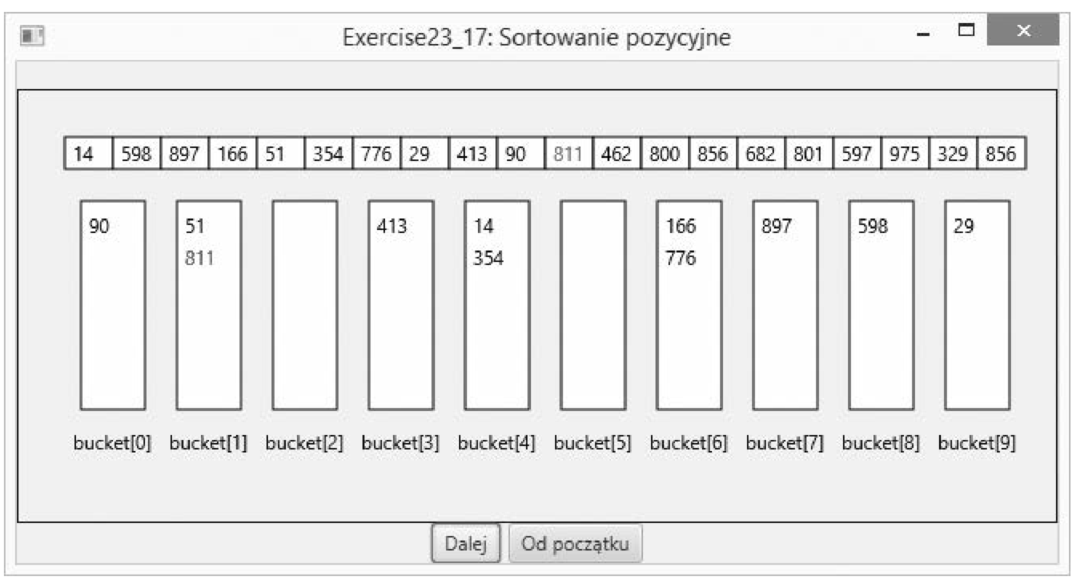
***23.16.** *Animacja sortowania bąbelkowego*. Napisz program wyświetlający animację sortowania bąbelkowego. Utwórz tablicę uporządkowanych losowo 20 różnych liczb od 1 do 20. Elementy tablicy należy wyświetlić w formie histogramu (rysunek 23.20b). Kliknięcie przycisku *Dalej* ma skutkować jednym porównaniem w algorytmie i ponownym wyświetleniem histogramu ze zmodyfikowaną tablicą. Wyróżnij kolorem sprawdzaną wartość. Po zakończeniu pracy algorytmu wyświetl informacje dla użytkownika. Przycisk *Od początku* ma tworzyć nową losową tablicę i umożliwić rozpoczęcie pracy od nowa.

Kup ksi k Pole ksi k



RYSUNEK 23.20. (a) Animacja sortowania przez wybieranie; (b) Animacja sortowania bąbelkowego

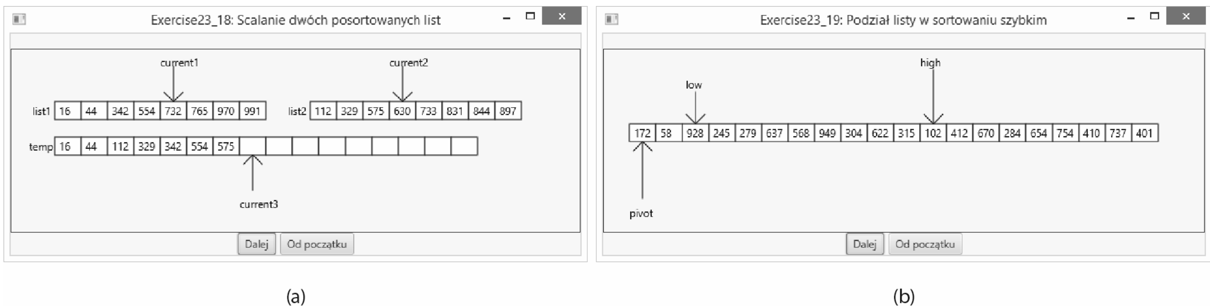
***23.17.** *Animacja sortowania pozycyjnego.* Napisz program wyświetlający animację sortowania pozycyjnego. Utwórz tablicę zawierającą 20 różnych liczb z przedziału od 0 do 1000. Elementy tablicy wyświetl w sposób pokazany na rysunku 23.21. Kliknięcie przycisku *Dalej* ma powodować umieszczenie liczby w kubełku. Ostatnią umieszczoną wartość należy wyróżnić kolorem czerwonym. Po umieszczeniu wszystkich liczb w kubełkach kliknięcie przycisku *Dalej* ma przenieść wszystkie wartości z kubełków z powrotem do tablicy. Po zakończeniu pracy algorytmu kliknięcie przycisku *Dalej* ma wyświetlać informacje dla użytkownika. Przycisk *Od początku* ma generować nową losową tablicę i umożliwiać rozpoczęcie pracy od początku.



RYSUNEK 23.21. Animacja sortowania pozycyjnego

***23.18.** *Animacja scalania.* Napisz program, który wyświetla animację scalania dwóch posortowanych list. Utwórz dwie tablice, `list1` i `list2`, z których każda zawiera osiem losowych liczb z przedziału od 1 do 999. Elementy tablicy wyświetl w sposób pokazany na rysunku 23.22a. Kliknięcie przycisku *Dalej* ma powodować przeniesienie elementu z `list1` lub `list2` do `temp`. Przycisk *Od początku* ma tworzyć dwie nowe losowe tablice i umożliwiać ponowne rozpoczęcie pracy. Po wykonaniu algorytmu przycisk *Dalej* ma wyświetlać informacje dla użytkownika.

***23.19.** *Animacja podziału w sortowaniu szybkim.* Napisz program, który wyświetla animację podziału w sortowaniu szybkim. Program ma tworzyć listę składającą się z 20 losowych liczb z przedziału od 1 do 999. Lista ma być wyświetlana w sposób pokazany na rysunku 23.22b. Przycisk *Dalej* ma powodować przesunięcie `low` w prawo lub `high` w lewo albo przestawienie elementów o indeksach `low` i `high`.



RYSUNEK 23.22. Animacja scalania dwóch posortowanych list; (b) Animacja podziału w sortowaniu szybkim

Kliknięcie przycisku *Od początku* ma tworzyć nową listę liczb losowych i umożliwić rozpoczęcie animacji od początku. Po zakończeniu pracy algorytmu przycisk *Dalej* ma powodować wyświetlenie informacji dla użytkownika.

Kup ksi k Pole ksi k

PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Mijają lata, dorastają kolejne pokolenia programistów, a Java wciąż zachwyca swoimi możliwościami. Jest językiem jednocześnie nowoczesnym, dojrzałym i… eleganckim. Twórcy Javy od jej pierwszego wydania starali się wdrażać awangardowe rozwiązania, pamiętając równocześnie o niezawodności i bezpieczeństwie kodu. Java wciąż pozostaje ulubionym narzędziem profesjonalistów tworzących aplikacje internetowe. Przy czym jej możliwości są o wiele większe. Aby jednak w przyszłości osiągać sukcesy jako programista Javy, trzeba gruntownie opanować podstawy tego języka i swobodnie operować instrukcjami sterującymi, pętlami, metodami i tablicami.

To dwunaste, rozszerzone wydanie znakomitego podręcznika do nauki Javy, w pełni przejrzane, poprawione i zaktualizowane (uwzględnia nowości wprowadzone w Javie SE 9, 10 i 11). Ułatwia zdobycie solidnych podstaw języka i płynne przejście do tworzenia programów służących do rozwiązywania konkretnych problemów z takich dziedzin jak matematyka, ekonomia, finanse, tworzenie gier i animacji. W książce precyzyjnie wyjaśniono zasady korzystania z różnych struktur danych i tworzenia algorytmów. Zamieszczono również wskazówki dotyczące ich implementacji i wydajności. Zrozumienie prezentowanych treści jest łatwiejsze dzięki licznym przykładom i ćwiczeniom do samodzielnego wykonania. Znakomitym uzupełnieniem materiału są także uwagi, ostrzeżenia i wskazówki programistyczne, zawierające cenne porady i przemyślenia.

W książce między innymi:

- solidne podstawy Javy
- programowanie zorientowane obiektowo
- projektowanie interfejsów użytkownika
- struktury danych i algorytmy
- wielowątkowość i programowanie równoległe

DR Y. DANIEL LIANG jest nagradzanym wykładowcą Uniwersytetu Purdue w Fort Wayne, autorem kilku książek o programowaniu i wielu artykułów publikowanych w periodykach naukowych. Specjalizuje się w projektowaniu algorytmów, przetwarzaniu typu klient – serwer oraz w zarządzaniu bazami danych.

Zostań profesjonalnym programistą Javy!

ISBN 978-83-283-7082-1
9 788328 370821
Cena: 199,00 zł