

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Ruby. Tao programowania w 400 przykładach

Autor: Hal Fulton

Tłumaczenie: Mikołaj Szczepaniak

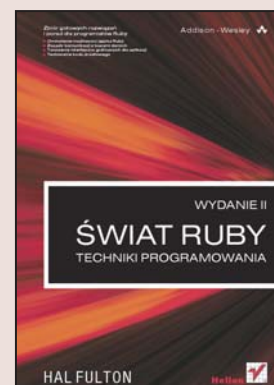
ISBN: 978-83-246-0958-1

Tytuł oryginału: [The Ruby Way, Second Edition: Solutions and Techniques in Ruby Programming](#)

Format: B5, stron: 912

oprawa twarda

[Przykłady na ftp: 54 kB](#)



Zbiór gotowych rozwiązań i porad dla programistów Ruby

- Omówienie możliwości języka Ruby
- Zasady komunikacji z bazami danych
- Tworzenie interfejsów graficznych dla aplikacji
- Testowanie kodu źródłowego

Ruby, obiektowy język programowania, opracowany na początku lat 90. ubiegłego wieku w Japonii, cieszy się zasłużoną i stale rosnącą popularnością. Dziś Ruby jest poważną konkurencją dla Perla i podstawowym fundamentem technologii Ruby on Rails – doskonałego narzędzia do szybkiego tworzenia aplikacji i witryn internetowych. Prosta składnia, duże możliwości, zwarta konstrukcja, rozbudowana i niezwykle wygodna obsługa wyjątków oraz przetwarzania plików tekstowych sprawiają, że po ten język programowania sięga coraz więcej osób piszących oprogramowanie.

Książka „Ruby. Tao programowania w 400 przykładach” to podręcznik dla tych programistów, którzy poszukują metod rozwiązywania konkretnych zadań programistycznych za pomocą Ruby. Na ponad 400 przykładach przedstawiono w niej przeróżne zastosowania i możliwości tego języka. Czytając ją, poznasz elementy języka Ruby i zasady programowania obiektowego, techniki przetwarzania łańcuchów tekstowych z zastosowaniem wyrażeń regularnych oraz sposoby wykonywania nawet najbardziej złożonych operacji matematycznych. Znajdziesz tu także omówienie metod komunikacji z bazami danych, budowania graficznych interfejsów użytkownika, programowania wielowątkowego i pisania skryptów administracyjnych. Dowiesz się też, jak korzystać z frameworka Ruby on Rails.

- Programowanie obiektowe w Ruby
- Przetwarzanie danych tekstowych
- Obliczenia matematyczne
- Internacjonalizacja aplikacji
- Operacje na złożonych strukturach danych
- Dynamiczne elementy języka Ruby
- Tworzenie interfejsów graficznych dla aplikacji
- Aplikacje wielowątkowe
- Pobieranie danych z baz
- Dystrybucja aplikacji
- Testowanie

**Tworzenie aplikacji internetowych w technologii Ruby on Rails
Przyspiesz proces tworzenia witryn i aplikacji z Ruby!**

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl



SPIS TREŚCI

	Słowo wstępne	21
	Podziękowania	25
	O autorze	29
	Wprowadzenie	31
ROZDZIAŁ 1.	Przegląd języka Ruby	47
	1.1. Wprowadzenie do programowania obiektowego	48
	1.1.1. Czym jest obiekt?	49
	1.1.2. Dziedziczenie	50
	1.1.3. Polimorfizm	53
	1.1.4. Kilka dodatkowych pojęć	54
	1.2. Podstawy składni i semantyki języka Ruby	55
	1.2.1. Słowa kluczowe i identyfikatory	57
	1.2.2. Komentarze i dokumentacja osadzana w kodzie źródłowym	58
	1.2.3. Stałe, zmienne i typy	58
	1.2.4. Operatory i priorytety operatorów	61
	1.2.5. Program przykładowy	62
	1.2.6. Pętle i struktury sterujące	65
	1.2.7. Wyjątki	70
	1.3. Programowanie obiektowe w języku Ruby	73
	1.3.1. Obiekty	74
	1.3.2. Klasy wbudowane	74
	1.3.3. Moduły i klasy mieszane	76
	1.3.4. Tworzenie klas	77
	1.3.5. Metody i atrybuty	82
	1.4. Aspekty dynamiczne języka programowania Ruby	84
	1.4.1. Kodowanie w czasie wykonywania	85
	1.4.2. Refleksja	86
	1.4.3. Brakujące metody	88
	1.4.4. Odzyskiwanie pamięci	89

1.5.	Ćwiczenie intuicji: o czym warto pamiętać	90
1.5.1.	Wybrane reguły składniowe	90
1.5.2.	Różne spojrzenia na programowanie	93
1.5.3.	Wyrażenie case języka Ruby	97
1.5.4.	Rubyizmy i idiomy	100
1.5.5.	Orientacja na wyrażenia i inne zagadnienia	106
1.6.	Żargon języka Ruby	108
1.7.	Konkluzja	112

ROZDZIAŁ 2.	Praca z łańcuchami	113
2.1.	Reprezentowanie typowych łańcuchów	114
2.2.	Reprezentowanie łańcuchów w notacjach alternatywnych	115
2.3.	Stosowanie dokumentu wbudowanego	115
2.4.	Określanie długości łańcuchów	118
2.5.	Przetwarzanie po jednym wierszu w każdej iteracji	118
2.6.	Przetwarzanie po jednym bajcie w każdej iteracji	118
2.7.	Stosowanie wyspecjalizowanych technik porównywania łańcuchów	119
2.8.	Dzielenie łańcuchów na tokeny	121
2.9.	Formatowanie łańcuchów	122
2.10.	Stosowanie łańcuchów w roli obiektów wejścia-wyjścia	123
2.11.	Konwersja wielkich i małych liter	123
2.12.	Uzyskiwanie dostępu i przypisywanie podłańcuchów	125
2.13.	Zamiana łańcuchów	127
2.14.	Przeszukiwanie łańcuchów	128
2.15.	Konwertowanie znaków na kody ASCII	129
2.16.	Konwersja jawna i niejawna	129
2.17.	Dołączanie elementów do łańcuchów	132
2.18.	Usuwanie końcowych znaków nowego wiersza i innych symboli specjalnych	133
2.19.	Usuwanie znaków białych z początku i końca łańcucha	134
2.20.	Powielanie łańcuchów	134
2.21.	Osadzanie wyrażeń w ramach łańcuchów	135
2.22.	Opóźnianie przetwarzania łańcuchów	135
2.23.	Analiza składniowa danych oddzielonych przecinkami	136
2.24.	Konwertowanie łańcuchów na liczby (dziesiętne i inne)	137
2.25.	Kodowanie i dekodowanie tekstu szyfrowanego za pomocą metody rot13	139
2.26.	Szyfrowanie łańcuchów	140
2.27.	Kompresja łańcuchów	141
2.28.	Wyznaczanie liczby wystąpień znaków w łańcuchach	142
2.29.	Odwracanie kolejności znaków w łańcuchu	142
2.30.	Usuwanie powtarzających się znaków	143
2.31.	Usuwanie określonych znaków	143
2.32.	Wyświetlanie znaków specjalnych	143

2.33.	Generowanie kolejnych łańcuchów	144
2.34.	Wyznaczanie 32-bitowych sum CRC	144
2.35.	Wyznaczanie kodów MD5 dla łańcuchów	145
2.36.	Wyznaczanie odległości Levenshteina dzielącej dwa łańcuchy	146
2.37.	Kodowanie i dekodowanie łańcuchów w formacie base64	148
2.38.	Kodowanie i dekodowanie łańcuchów za pomocą narzędzi uuencode oraz uudecode	149
2.39.	Rozszerzanie i kompresja znaków tabulacji	149
2.40.	Opakowywanie wierszy tekstu	150
2.41.	Konkluzja	151
ROZDZIAŁ 3.	Praca z wyrażeniami regularnymi	153
3.1.	Składnia wyrażeń regularnych	154
3.2.	Kompilowanie wyrażeń regularnych	156
3.3.	Stosowanie znaków specjalnych	157
3.4.	Stosowanie tzw. kotwic	157
3.5.	Stosowanie kwantyfikatorów	158
3.6.	Antycypacja dodatnia i ujemna	160
3.7.	Uzyskiwanie dostępu do referencji wstecznych	161
3.8.	Stosowanie klas znaków	165
3.9.	Rozszerzone wyrażenia regularne	166
3.10.	Dopasowywanie znaku nowego wiersza do kropki	167
3.11.	Stosowanie opcji osadzanych	168
3.12.	Stosowanie podwyrażeń osadzanych	169
3.13.	Ruby i Oniguruma	169
3.13.1.	Testowanie dostępności mechanizmu Oniguruma	170
3.13.2.	Kompilacja silnika Oniguruma	171
3.13.3.	Przegląd wybranych nowości zaimplementowanych w silniku Oniguruma	172
3.13.4.	Dodatnia i ujemna antycypacja wsteczna	172
3.13.5.	Więcej o kwantyfikatorach	174
3.13.6.	Dopasowania nazwane	174
3.13.7.	Rekurencja w wyrażeniach regularnych	176
3.14.	Kilka przykładowych wyrażeń regularnych	177
3.14.1.	Dopasowywanie adresów IP	177
3.14.2.	Dopasowywanie par klucz-wartość	178
3.14.3.	Dopasowywanie liczb rzymskich	179
3.14.4.	Dopasowywanie stałych numerycznych	179
3.14.5.	Dopasowywanie łańcuchów zawierających datę i godzinę	180
3.14.6.	Wykrywanie powtarzających się wyrazów w tekście	181
3.14.7.	Dopasowywanie słów pisanych wielkimi literami	181
3.14.8.	Dopasowywanie numerów wersji	182
3.14.9.	Kilka dodatkowych wzorców	182
3.15.	Konkluzja	183

ROZDZIAŁ 4.	Umieędzynaradawianie aplikacji Ruby	185
4.1.	Wstęp teoretyczny i terminologia	187
4.2.	Kodowanie znaków we współczesnym świecie (po rezygnacji ze standardu ASCII)	191
4.2.1.	Biblioteka jcode i zmienna globalna \$KCODE	192
4.2.2.	Ponowne spojrzenie na popularne operacje na łańcuchach i wyrażeniach regularnych	193
4.2.3.	Wykrywanie schematów kodowania znaków	198
4.2.4.	Normalizacja łańcuchów Unicode	198
4.2.5.	Problemy związane z porządkowaniem łańcuchów	200
4.2.6.	Konwertowanie łańcuchów zakodowanych według różnych schematów	204
4.3.	Stosowanie katalogów komunikatów	207
4.3.1.	Wstęp teoretyczny i terminologia	207
4.3.2.	Pierwsze kroki w świecie katalogów komunikatów	208
4.3.3.	Lokalizacja prostej aplikacji	209
4.3.4.	Informacje dodatkowe	214
4.4.	Konkluzja	215
ROZDZIAŁ 5.	Wykonywanie obliczeń numerycznych	217
5.1.	Reprezentowanie liczb w języku Ruby	218
5.2.	Podstawowe operacje na liczbach	219
5.3.	Zaokrąglanie liczb zmiennoprzecinkowych	220
5.4.	Porównywanie liczb zmiennoprzecinkowych	222
5.5.	Formatowanie liczb przeznaczonych do wyświetlenia	223
5.6.	Formatowanie liczb z separatorami tysięcy	224
5.7.	Praca z bardzo dużymi liczbami całkowitymi	225
5.8.	Stosowanie typu BigDecimal	225
5.9.	Praca z liczbami wymiernymi	227
5.10.	Operacje na macierzach	228
5.11.	Praca z liczbami zespolonymi	233
5.12.	Stosowanie biblioteki mathn	234
5.13.	Rozkład na czynniki pierwsze, największy wspólny dzielnik i najmniejsza wspólna wielokrotność	235
5.14.	Praca z liczbami pierwszymi	236
5.15.	Niejawna i bezpośrednia konwersja numeryczna	237
5.16.	Koercja wartości numerycznych	238
5.17.	Wykonywanie operacji bitowych na liczbach	240
5.18.	Konwersje pomiędzy systemami liczbowymi	241
5.19.	Wyznaczanie pierwiastków sześciennych, czwartego stopnia, piątego stopnia itd.	242
5.20.	Określanie porządku bajtów obowiązującego w danej architekturze	243
5.21.	Numeryczna metoda wyznaczania całki oznaczonej	244
5.22.	Trygonometria w stopniach, radianach i gradach	245

5.23.	Bardziej zaawansowane funkcje trygonometryczne	247
5.24.	Wyznaczanie logarytmów o dowolnych podstawach	247
5.25.	Wyznaczanie wartości średniej, mediany i mody zbioru danych	248
5.26.	Wariancja i odchylenie standardowe	249
5.27.	Wyznaczanie współczynnika korelacji	250
5.28.	Generowanie liczb losowych	251
5.29.	Składowanie wyników funkcji w pamięci podręcznej za pomocą biblioteki memoize	252
5.30.	Konkluzja	254
ROZDZIAŁ 6.	Symbole i przedziały	255
6.1.	Symbole	256
6.1.1.	Symbole jako typy wyliczeniowe	258
6.1.2.	Symbole jako metawartości	258
6.1.3.	Symbole, zmienne i metody	259
6.1.4.	Konwertowanie na symbole i z symboli	260
6.2.	Przedziały	261
6.2.1.	Przedziały otwarte i domknięte	262
6.2.2.	Wyznaczanie punktów końcowych	262
6.2.3.	Iteracyjne przeszukiwanie przedziałów	263
6.2.4.	Sprawdzanie przynależności do przedziałów	264
6.2.5.	Konwertowanie przedziałów na tablice	264
6.2.6.	Przedziały odwrotne	265
6.2.7.	Operator przerzutnikowy	265
6.2.8.	Przedziały niestandardowe	269
6.3.	Konkluzja	272
ROZDZIAŁ 7.	Praca z datami i godzinami	273
7.1.	Określanie bieżącej godziny	274
7.2.	Praca z określonymi datami i godzinami (począwszy od punktu nazywanego epoką)	275
7.3.	Określanie dnia tygodnia	276
7.4.	Określanie daty Wielkanocy	277
7.5.	Określanie daty n-tego dnia tygodnia w danym miesiącu	277
7.6.	Konwersja pomiędzy sekundami a większymi jednostkami czasu	279
7.7.	Konwersja daty i godziny do postaci i z postaci epoki	280
7.8.	Praca z sekundami przestępnymi — nie róbcie tego w domu!	280
7.9.	Wyznaczanie numeru dnia w danym roku	281
7.10.	Sprawdzanie poprawności daty i godziny	281
7.11.	Określanie numeru tygodnia w danym roku	283
7.12.	Wykrywanie roku przestępnego	284
7.13.	Określanie strefy czasowej	285

7.14.	Praca z samymi godzinami i minutami	285
7.15.	Porównywanie wartości reprezentujących daty i godziny	285
7.16.	Dodawanie i odejmowanie przedziałów czasowych do i od wartości reprezentujących daty i godziny	286
7.17.	Wyznaczanie różnic dzielących dwie wartości reprezentujące daty i godziny	287
7.18.	Praca z określonymi datami i godzinami (sprzed punktu nazywanego epoką)	287
7.19.	Wzajemna konwersja obiektów klasy Time, Date oraz DateTime	288
7.20.	Odczytywanie daty i godziny z łańcucha wejściowego	289
7.21.	Formatowanie i wyświetlanie daty i godziny	291
7.22.	Konwersja stref czasowych	292
7.23.	Określanie liczby dni danego miesiąca	292
7.24.	Dzielenie miesiąca na tygodnie	293
7.25.	Konkluzja	294

ROZDZIAŁ 8. Tablice, tablice mieszające i inne wyliczeniowe struktury danych 295

8.1.	Praca z tablicami	296
8.1.1.	Tworzenie i inicjalizacja tablic	296
8.1.2.	Uzyskiwanie dostępu i przypisywanie wartości elementom tablicy	297
8.1.3.	Określanie rozmiaru tablicy	299
8.1.4.	Porównywanie tablic	299
8.1.5.	Sortowanie elementów tablicy	301
8.1.6.	Selekcja elementów tablicy według określonych kryteriów	304
8.1.7.	Stosowanie wyspecjalizowanych funkcji indeksujących	306
8.1.8.	Implementacja macierzy rzadkich	308
8.1.9.	Stosowanie tablic w roli zbiorów matematycznych	309
8.1.10.	Losowe porządkowanie elementów tablicy	313
8.1.11.	Stosowanie tablic wielowymiarowych	314
8.1.12.	Identyfikacja tych elementów jednej tablicy, które nie występują w innej tablicy	315
8.1.13.	Transformowanie i odwzorowywanie tablic	315
8.1.14.	Usuwanie wartości nil z tablicy	316
8.1.15.	Usuwanie określonych elementów tablicy	316
8.1.16.	Konkatenacja i dołączanie tablic	317
8.1.17.	Stosowanie tablic w roli stosów i kolejek	318
8.1.18.	Iteracyjne przeszukiwanie tablic	319
8.1.19.	Wstawianie separatorów uwzględnianych w łańcuchu wynikowym	320
8.1.20.	Odwracanie kolejności elementów tablicy	320
8.1.21.	Usuwanie z tablicy powtarzających się elementów	320

8.1.22.	Przeplatanie tablic	321
8.1.23.	Zliczanie częstotliwości występowania poszczególnych wartości w tablicy	321
8.1.24.	Odwracanie kierunku relacji w tablicy przez tworzenie odpowiedniej tablicy mieszającej	321
8.1.25.	Zsynchronizowane sortowanie wielu tablic	322
8.1.26.	Określanie wartości domyślnej dla nowych elementów tablicy	323
8.2.	Praca z tablicami mieszającymi	324
8.2.1.	Tworzenie nowych tablic mieszających	324
8.2.2.	Określanie wartości domyślnej dla tablicy mieszającej	325
8.2.3.	Uzyskiwanie dostępu i dodawanie par klucz-wartość	326
8.2.4.	Usuwanie par klucz-wartość	327
8.2.5.	Iteracyjne przeszukiwanie tablicy mieszającej	328
8.2.6.	Odwracanie związków w tablicy mieszającej	328
8.2.7.	Wykrywanie kluczy i wartości w tablicy mieszającej	329
8.2.8.	Konwersja tablic mieszających na tablice	329
8.2.9.	Wyodrębnianie par klucz-wartość według określonych kryteriów	330
8.2.10.	Sortowanie tablicy mieszającej	330
8.2.11.	Scalanie dwóch tablic mieszających	331
8.2.12.	Tworzenie tablic mieszających na podstawie tablic	331
8.2.13.	Wyznaczanie różnicy i iloczynu (części wspólnej) kluczy zbioru tablic mieszających	331
8.2.14.	Stosowanie tablic mieszających w roli reprezentacji macierzy rzadkich	332
8.2.15.	Implementacja tablic mieszających obsługujących powtarzające się klucze	333
8.3.	Ogólne omówienie typów wyczerpieniowych	336
8.3.1.	Metoda inject	337
8.3.2.	Stosowanie kwalifikatorów	338
8.3.3.	Metoda partition	339
8.3.4.	Iteracyjne przeszukiwanie kolekcji grupami elementów	340
8.3.5.	Konwersja tablic na zbiory	341
8.3.6.	Stosowanie obiektów klasy Enumerator	341
8.3.7.	Stosowanie obiektów klasy Generator	343
8.4.	Konkluzja	344
ROZDZIAŁ 9.	Zaawansowane struktury danych	347
9.1.	Praca ze zbiorami	348
9.1.1.	Proste operacje na zbiorach	348
9.1.2.	Zaawansowane operacje na zbiorach	350

9.2.	Praca ze stosami i kolejkami	351
9.2.1.	Implementacja stosu wymuszającego właściwy dostęp do danych	353
9.2.2.	Wykrywanie niezbilansowanych znaków interpunkcyjnych w wyrażeniach	354
9.2.3.	Stosy i rekurencja	355
9.2.4.	Implementacja kolejki wymuszającej właściwy dostęp do danych	357
9.3.	Praca z drzewami	358
9.3.1.	Implementacja drzewa binarnego	359
9.3.2.	Sortowanie danych z wykorzystaniem drzewa binarnego	361
9.3.3.	Stosowanie drzewa binarnego w roli tablicy wyszukiwania	363
9.3.4.	Konwersja drzewa na łańcuch lub tablicę	364
9.4.	Praca z grafami	365
9.4.1.	Implementacja grafu w formie macierzy sąsiedztwa	366
9.4.2.	Określanie, czy wszystkie węzły grafu są z nim połączone	368
9.4.3.	Określanie, czy dany graf zawiera cykl Eulera	370
9.4.4.	Określanie, czy dany graf zawiera ścieżkę Eulera	371
9.4.5.	Narzędzia ułatwiające operacje na grafach w języku Ruby	371
9.5.	Konkluzja	372

ROZDZIAŁ 10.	Operacje wejścia-wyjścia i techniki składowania danych	373
10.1.	Praca z plikami i katalogami	375
10.1.1.	Otwieranie i zamykanie plików	375
10.1.2.	Aktualizacja pliku	377
10.1.3.	Dopisywanie danych do istniejącego pliku	377
10.1.4.	Swobodny dostęp do zawartości plików	377
10.1.5.	Praca z plikami binarnymi	378
10.1.6.	Blokowanie dostępu do plików	380
10.1.7.	Wykonywanie prostych operacji wejścia-wyjścia	381
10.1.8.	Wykonywanie buforowanych i niebuforowanych operacji wejścia-wyjścia	382
10.1.9.	Modyfikowanie uprawnień dostępu i praw własności do plików	383
10.1.10.	Uzyskiwanie i ustawianie informacji o znacznikach czasowych	385
10.1.11.	Weryfikacja istnienia i rozmiaru pliku	387
10.1.12.	Weryfikacja specjalnych charakterystyk plików	388
10.1.13.	Praca z potokami	390
10.1.14.	Wykonywanie specjalnych operacji wejścia-wyjścia	392
10.1.15.	Stosowanie nieblokujących operacji wejścia-wyjścia	393
10.1.16.	Stosowanie metody readpartial	393

10.1.17.	Modyfikowanie ścieżek do plików	394
10.1.18.	Stosowanie klasy Pathname	395
10.1.19.	Wykonywanie operacji na plikach na poziomie poleceń	396
10.1.20.	Przechwytywanie znaków z klawiatury	398
10.1.21.	Odczytywanie i umieszczanie w pamięci całych plików	399
10.1.22.	Iteracyjne przeszukiwanie pliku wejściowego wiersz po wierszu	399
10.1.23.	Iteracyjne przeszukiwanie pliku wejściowego bajt po bajcie	400
10.1.24.	Traktowanie łańcuchów jak plików	400
10.1.25.	Odczytywanie danych osadzonych w kodzie źródłowym programu	401
10.1.26.	Odczytywanie kodu źródłowego programu	401
10.1.27.	Praca z plikami tymczasowymi	402
10.1.28.	Zmianianie i ustawianie katalogu bieżącego	403
10.1.29.	Zmiana bieżącego katalogu głównego	403
10.1.30.	Iteracyjne przeszukiwanie listy plików i podkatalogów	404
10.1.31.	Uzyskiwanie listy plików i podkatalogów	404
10.1.32.	Tworzenie łańcucha katalogów	404
10.1.33.	Rekurencyjne usuwanie katalogów	405
10.1.34.	Odnajdywanie plików i katalogów	405
10.2.	Uzyskiwanie dostępu do danych na wyższym poziomie	406
10.2.1.	Proste utrwalanie obiektów	406
10.2.2.	Bardziej złożone utrwalanie obiektów	408
10.2.3.	Sporządzanie „głębokiej kopii” w ograniczonej formie	409
10.2.4.	Udoskonalone utrwalanie obiektów za pomocą biblioteki PStore	409
10.2.5.	Praca z danymi CSV	411
10.2.6.	Utrwalanie danych z wykorzystaniem formatu YAML	413
10.2.7.	Przezroczysta architektura utrwalania obiektów za pomocą projektu Madeleine	414
10.2.8.	Stosowanie biblioteki DBM	415
10.3.	Stosowanie biblioteki KirbyBase	417
10.4.	Nawiązywanie połączeń z zewnętrznymi bazami danych	420
10.4.1.	Korzystanie z interfejsu bazy danych SQLite	421
10.4.2.	Korzystanie z interfejsu bazy danych MySQL	422
10.4.3.	Korzystanie z interfejsu bazy danych PostgreSQL	425
10.4.4.	Korzystanie z interfejsu do protokołu LDAP	429
10.4.5.	Korzystanie z interfejsu bazy danych Oracle	430

10.4.6.	Stosowanie opakowania DBI	432
10.4.7.	Mechanizmy odwzorowań obiektowo-relacyjnych	433
10.5.	Konkluzja	435
ROZDZIAŁ 11.	Programowanie obiektowe i dynamiczne elementy języka Ruby	437
11.1.	Programowanie obiektowe w codziennej pracy	438
11.1.1.	Stosowanie wielu konstruktorów	439
11.1.2.	Tworzenie atrybutów egzemplarzy	440
11.1.3.	Stosowanie bardziej złożonych konstruktorów	441
11.1.4.	Tworzenie atrybutów i metod na poziomie klas	443
11.1.5.	Dziedziczenie po nadklasie	447
11.1.6.	Testowanie klas obiektów	449
11.1.7.	Testowanie równości obiektów	452
11.1.8.	Kontrola dostępu do metod	453
11.1.9.	Kopiowanie obiektów	455
11.1.10.	Stosowanie metody initialize_copy	457
11.1.11.	Wyjaśnienie znaczenia metody allocate	458
11.1.12.	Praca z modułami	459
11.1.13.	Transformowanie i konwertowanie obiektów	462
11.1.14.	Tworzenie klas (struktur) zawierających wyłącznie dane	466
11.1.15.	Zamrażanie obiektów	467
11.2.	Techniki zaawansowane	469
11.2.1.	Wysyłanie obiektom komunikatów wyrażonych wprost	469
11.2.2.	Specjalizacja pojedynczych obiektów	471
11.2.3.	Zagnieżdżanie klas i modułów	475
11.2.4.	Tworzenie klas parametrycznych	476
11.2.5.	Stosowanie kontynuacji w implementacji generatora	479
11.2.6.	Składowanie kodu w formie obiektów	481
11.2.7.	Omówienie mechanizmu zawierania modułów	483
11.2.8.	Wykrywanie parametrów domyślnych	485
11.2.9.	Delegowanie wywołań i przekazywanie ich dalej	486
11.2.10.	Automatyczne definiowanie metod odczytujących i zapisujących na poziomie klasy	488
11.2.11.	Stosowanie zaawansowanych technik programistycznych	490
11.3.	Praca z elementami dynamicznymi języka Ruby	493
11.3.1.	Dynamiczne przetwarzanie kodu	494
11.3.2.	Stosowanie metody const_get	495
11.3.3.	Dynamiczne tworzenie egzemplarzy klasy reprezentowanej przez nazwę	496
11.3.4.	Zwracanie i ustawianie zmiennych egzemplarzy	497
11.3.5.	Stosowanie wyrażenia define_method	498

11.3.6.	Stosowanie metody <code>const_missing</code>	502
11.3.7.	Usuwanie definicji	503
11.3.8.	Generowanie list zdefiniowanych konstrukcji	505
11.3.9.	Analiza stosu wywołań	507
11.3.10.	Monitorowanie wykonywania programu	508
11.3.11.	Przeszukiwanie przestrzeni obiektów	510
11.3.12.	Obsługa wywołań nieistniejących metod	510
11.3.13.	Śledzenie zmian w definicji klasy lub obiektu	511
11.3.14.	Definiowanie finalizatorów obiektów	515
11.4.	Konkluzja	517
ROZDZIAŁ 12.	Interfejsy graficzne dla Ruby	519
12.1.	Ruby i Tk	520
12.1.1.	Wprowadzenie	521
12.1.2.	Prosta aplikacja okienkowa	522
12.1.3.	Praca z przyciskami	524
12.1.4.	Praca z polami tekstowymi	528
12.1.5.	Praca z pozostałymi rodzajami kontrolek	532
12.1.6.	Informacje dodatkowe	536
12.2.	Ruby i GTK2	537
12.2.1.	Wprowadzenie	537
12.2.2.	Prosta aplikacja okienkowa	538
12.2.3.	Praca z przyciskami	540
12.2.4.	Praca z polami tekstowymi	542
12.2.5.	Praca z pozostałymi rodzajami kontrolek	545
12.2.6.	Informacje dodatkowe	550
12.3.	FXRuby (FOX)	553
12.3.1.	Wprowadzenie	553
12.3.2.	Prosta aplikacja okienkowa	555
12.3.3.	Praca z przyciskami	556
12.3.4.	Praca z polami tekstowymi	558
12.3.5.	Praca z pozostałymi rodzajami kontrolek	560
12.3.6.	Informacje dodatkowe	569
12.4.	QtRuby	570
12.4.1.	Wprowadzenie	570
12.4.2.	Prosta aplikacja okienkowa	571
12.4.3.	Praca z przyciskami	572
12.4.4.	Praca z polami tekstowymi	574
12.4.5.	Praca z pozostałymi rodzajami kontrolek	576
12.4.6.	Informacje dodatkowe	581
12.5.	Pozostałe zestawy narzędzi GUI	582
12.5.1.	Ruby i środowisko X	582
12.5.2.	Ruby i system wxWidgets	583
12.5.3.	Apollo (Ruby i Delphi)	583
12.5.4.	Ruby i interfejs Windows API	584
12.6.	Konkluzja	584

ROZDZIAŁ 13. Wątki w języku Ruby	585
13.1. Tworzenie wątków i zarządzanie nimi	586
13.1.1. Tworzenie wątków	587
13.1.2. Uzyskiwanie dostępu do zmiennych lokalnych wątków	588
13.1.3. Sprawdzanie i modyfikowanie stanu wątku	590
13.1.4. Oczekiwanie na wątek potomny (i przechwytywanie zwracanej wartości)	593
13.1.5. Obsługa wyjątków	595
13.1.6. Stosowanie grup wątków	596
13.2. Synchronizacja wątków	597
13.2.1. Proste synchronizowanie wątków z wykorzystaniem sekcji krytycznych	599
13.2.2. Synchronizacja dostępu do zasobów (biblioteka mutex.rb)	600
13.2.3. Stosowanie predefiniowanych klas kolejek synchronizowanych	604
13.2.4. Stosowanie zmiennych warunkowych	605
13.2.5. Stosowanie pozostałych technik synchronizacji	607
13.2.6. Stosowanie limitów czasowych dla operacji	610
13.2.7. Oczekiwanie na zdarzenia	611
13.2.8. Kontynuacja przetwarzania w czasie wykonywania operacji wejścia-wyjścia	612
13.2.9. Implementacja iteratorów równoległych	613
13.2.10. Rekurencyjne, równoległe usuwanie plików i katalogów	615
13.3. Konkluzja	616
ROZDZIAŁ 14. Tworzenie skryptów i administracja systemem	617
14.1. Uruchamianie programów zewnętrznych	618
14.1.1. Stosowanie metod system i exec	618
14.1.2. Przechwytywanie danych wyjściowych wykonywanego polecenia	620
14.1.3. Operacje na procesach	621
14.1.4. Operacje na standardowym wejściu-wyjściu	624
14.2. Opcje i argumenty wiersza poleceń	624
14.2.1. Analiza składniowa opcji wiersza poleceń	625
14.2.2. Stała ARGF	627
14.2.3. Stała ARGV	628
14.3. Biblioteka Shell	629
14.3.1. Przekierowywanie wejścia-wyjścia za pomocą klasy Shell	629
14.3.2. Informacje dodatkowe o bibliotece shell.rb	631

14.4.	Uzyskiwanie dostępu do zmiennych środowiskowych	632
14.4.1.	Odczytywanie i ustawianie wartości zmiennych środowiskowych	632
14.4.2.	Składowanie zmiennych środowiskowych w formie tablic lub tablic mieszających	633
14.4.3.	Importowanie zmiennych środowiskowych do postaci zmiennych globalnych aplikacji	634
14.5.	Wykonywanie skryptów w systemach Microsoft Windows	635
14.5.1.	Stosowanie biblioteki Win32API	636
14.5.2.	Stosowanie biblioteki Win32OLE	637
14.5.3.	Stosowanie interfejsu ActiveScriptRuby	640
14.6.	Wygodny instalator dla systemu Windows	641
14.7.	Biblioteki, które warto znać	643
14.8.	Praca z plikami, katalogami i drzewami	644
14.8.1.	Kilka słów o filtrach tekstu	644
14.8.2.	Kopiowanie drzewa katalogów (obejmującego dowiązania symetryczne)	645
14.8.3.	Usuwanie plików według wieku i innych kryteriów	647
14.8.4.	Określanie ilości wolnej przestrzeni na dysku	648
14.9.	Różne zadania realizowane za pomocą skryptów	648
14.9.1.	Programy języka Ruby w formie pojedynczych plików	649
14.9.2.	Kierowanie potoku do interpretera języka Ruby	650
14.9.3.	Uzyskiwanie i ustawianie kodów wyjścia	651
14.9.4.	Sprawdzanie, czy dany program pracuje w trybie interaktywnym	652
14.9.5.	Określanie bieżącej platformy lub systemu operacyjnego	652
14.9.6.	Stosowanie modułu Etc	653
14.10.	Konkluzja	654
ROZDZIAŁ 15.	Ruby i formaty danych	655
15.1.	Analiza składniowa danych w formacie XML za pomocą biblioteki REXML	656
15.1.1.	Analiza składniowa z wykorzystaniem struktury drzewa	658
15.1.2.	Analiza składniowa danych w formie strumienia	659
15.1.3.	Język XPath i inne	660
15.2.	Praca z formatami RSS i Atom	661
15.2.1.	Biblioteka standardowa rss	661
15.2.2.	Biblioteka feedtools	664
15.3.	Operowanie na obrazach za pośrednictwem biblioteki RMagick	666
15.3.1.	Typowe operacje na obrazach	667
15.3.2.	Przekształcenia i efekty specjalne	670
15.3.3.	Interfejs API umożliwiający rysowanie	672

15.4.	Tworzenie dokumentów PDF za pomocą biblioteki PDF::Writer	677
15.4.1.	Podstawowe pojęcia i techniki	677
15.4.2.	Dokument przykładowy	679
15.5.	Konkluzja	687
ROZDZIAŁ 16.	Testowanie i diagnozowanie oprogramowania	689
16.1.	Testowanie aplikacji za pomocą biblioteki Test::Unit	690
16.2.	Narzędzia ZenTest	695
16.3.	Stosowanie debugera języka Ruby	698
16.4.	Stosowanie narzędzia irb w roli debugera programów napisanych w Ruby	701
16.5.	Ocena pokrycia kodu testami	703
16.6.	Ocena wydajności	704
16.7.	Stosowanie techniki pretty-printing dla obiektów	709
16.8.	Konkluzja	711
ROZDZIAŁ 17.	Pakowanie i dystrybucja kodu źródłowego	713
17.1.	Stosowanie narzędzia RDoc	714
17.1.1.	Stosowanie znaczników formatujących	715
17.1.2.	Bardziej zaawansowane techniki formatowania	719
17.2.	Instalacja i pakowanie	720
17.2.1.	Biblioteka setup.rb	720
17.2.2.	System RubyGems	723
17.3.	Projekty RubyForge i RAA	724
17.4.	Konkluzja	727
ROZDZIAŁ 18.	Programowanie rozwiązań sieciowych	729
18.1.	Serwery sieciowe	731
18.1.1.	Prosty serwer — która godzina?	732
18.1.2.	Implementacja serwera wielowątkowego	734
18.1.3.	Studium przypadku: serwer szachowy pracujący w trybie równorzędnym	734
18.2.	Aplikacje klienckie	743
18.2.1.	Uzyskiwanie za pośrednictwem internetu prawdziwych liczb losowych	744
18.2.2.	Nawiązywanie połączeń z oficjalnym serwerem czasu	747
18.2.3.	Komunikacja z serwerem POP	748
18.2.4.	Wysyłanie wiadomości poczty elektronicznej za pośrednictwem protokołu SMTP	750
18.2.5.	Komunikacja z serwerem IMAP	753
18.2.6.	Kodowanie i dekodowanie załączników	756
18.2.7.	Studium przypadku: brama łącząca listę dyskusyjną z grupą dyskusyjną	758

18.2.8.	Uzyskiwanie strony internetowej na podstawie adresu URL	763
18.2.9.	Stosowanie biblioteki Open-URI	764
18.3.	Konkluzja	765
ROZDZIAŁ 19.	Ruby i aplikacje internetowe	767
19.1.	Programowanie w języku Ruby aplikacji CGI	767
19.1.1.	Wprowadzenie do biblioteki cgi.rb	769
19.1.2.	Wyświetlanie i przetwarzanie formularzy	771
19.1.3.	Praca ze znacznikami kontekstu klienta	772
19.1.4.	Praca z sesjami użytkownika	773
19.2.	Stosowanie technologii FastCGI	774
19.3.	Framework Ruby on Rails	776
19.3.1.	Podstawowe zasady i techniki programowania	777
19.3.2.	Testowanie i diagnozowanie aplikacji budowanych na bazie frameworku Rails	779
19.3.3.	Rozszerzenia zbioru klas podstawowych	780
19.3.4.	Narzędzia i biblioteki pokrewne	781
19.4.	Wytwarzanie aplikacji internetowych z wykorzystaniem zestawu narzędzi Nitro	782
19.4.1.	Tworzenie prostych aplikacji na bazie zestawu narzędzi Nitro	783
19.4.2.	Zestaw narzędzi Nitro i wzorzec projektowy MVC	785
19.4.3.	Nitro i mechanizm Og	790
19.4.4.	Realizacja typowych zadań budowy aplikacji internetowych z wykorzystaniem zestawu narzędzi Nitro	791
19.4.5.	Informacje dodatkowe	795
19.5.	Wprowadzenie do frameworku Wee	797
19.5.1.	Prosty przykład	798
19.5.2.	Wiązanie stanu z adresami URL	799
19.6.	Wytwarzanie aplikacji internetowych z wykorzystaniem frameworku IOWA	801
19.6.1.	Podstawowe cechy frameworku IOWA	801
19.6.2.	Stosowanie szablonów w aplikacjach budowanych na bazie frameworku IOWA	804
19.6.3.	Transfer sterowania pomiędzy komponentami	805
19.7.	Ruby i serwery WWW	807
19.7.1.	Stosowanie modułu mod_ruby	808
19.7.2.	Stosowanie narzędzia erb	809
19.7.3.	Stosowanie serwera WEBrick	812
19.7.4.	Stosowanie serwera Mongrel	814
19.8.	Konkluzja	817

ROZDZIAŁ 20. Implementacja aplikacji rozproszonych w Ruby	819
20.1. Wprowadzenie do biblioteki drb	820
20.2. Studium przypadku: symulacja systemu publikacji aktualnych notowań papierów wartościowych	823
20.3. Biblioteka Rinda: przestrzeń krotek języka Ruby	827
20.4. Wyszukiwanie usług za pomocą mechanizmów rozproszonych języka Ruby	832
20.5. Konkluzja	833
ROZDZIAŁ 21. Narzędzia wytwarzania oprogramowania w Ruby	835
21.1. Stosowanie systemu RubyGems	836
21.2. Narzędzie rake	838
21.3. Stosowanie narzędzia irb	843
21.4. Narzędzie ri	848
21.5. Edytory kodu	849
21.6. Zintegrowane środowiska wytwarzania	851
21.7. Konkluzja	852
ROZDZIAŁ 22. Społeczność programistów języka Ruby	855
22.1. Zasoby internetowe	855
22.2. Grupy i listy dyskusyjne	856
22.3. Blogi i czasopisma internetowe	857
22.4. Dokumenty RCR	857
22.5. Kanały IRC	858
22.6. Konferencje poświęcone programowaniu w Ruby	859
22.7. Lokalne grupy programistów Ruby	860
22.8. Konkluzja	860
Skorowidz	861

ROZDZIAŁ 9.

Zaawansowane struktury danych

Graficzne odwzorowanie danych pobieranych z banków wszystkich komputerów świata. Niewyobrazalna złożoność... Linie światła sięgającego nieprzestrzeni umysłu, roje i konstelacje danych. Jak światła miasta, milknące w oddali.

— William Gibson

Istnieją oczywiście bardziej złożone i interesujące struktury danych niż tablice, tablice mieszające i pokrewne. Część spośród struktur danych, którymi zajmiemy się w niniejszym rozdziale, jest pośrednio lub bezpośrednio obsługiwana w języku Ruby; pozostałe będziemy musieli w większym lub mniejszym stopniu implementować samodzielnie. Na szczęście okazuje się, że język programowania Ruby znacznie upraszcza proces konstruowania niestandardowych struktur danych.

Jak się za chwilę okaże, zbiór matematyczny może być reprezentowany w formie tablic. Najnowsza wersja języka Ruby dodatkowo oferuje swoim programistom klasę `Set`, która doskonale sprawdza się w tej roli.

Stosy i kolejki należą do najbardziej popularnych struktur danych w świecie komputerów. Czytelnicy zainteresowani praktycznymi zastosowaniami tych struktur danych znajdą kilka przykładów w niniejszym rozdziale. Bardziej szczegółowe omówienie zagadnień związanych ze stosami i kolejkami można znaleźć w niezliczonych podręcznikach przeznaczonych dla studentów pierwszego roku informatyki.

Drzewa są szczególnie przydatne w procesach sortowania, przeszukiwania i zwykłego reprezentowania danych hierarchicznych. W tym rozdziale skoncentrujemy się przede wszystkim na drzewach binarnych, ale zajmiemy się też innymi strukturami tego typu.

Bardziej ogólną formą drzewa jest **graf** (ang. *graph*). Graf jest po prostu kolekcją węzłów połączonych łukami, które mogą mieć przypisane wagi i kierunki. Grafy okazują się szczególnie przydatne w takich obszarach badań jak sieci komputerowe czy inżynieria wiedzy.

Ponieważ jednak najprostszym zagadnieniem są zbiory, właśnie nimi zajmiemy się w pierwszej kolejności.

9.1. PRACA ZE ZBIORAMI

W poprzednim rozdziale mieliśmy okazję się przekonać, jak pewne metody klasy `Array` umożliwiają nam stosowanie jej obiektów w roli możliwej do przyjęcia reprezentacji zbiorów. Nieco większe możliwości i wygodę w tym obszarze oferuje klasa `Set`, która ukrywa przed programistą część szczegółów implementacyjnych.

Aby mieć dostęp do rozwiązań zaimplementowanych w klasie `Set`, wystarczy użyć prostego wyrażenia `require`:

```
require 'set'
```

W ten sposób dodatkowo rozbudowujemy moduł `Enumerable` o metodę `to_set`, która umożliwia konwersję na zbiór dowolnych obiektów reprezentujących wyliczeniowe struktury danych.

Tworzenie nowych zbiorów jest bardzo proste. Okazuje się, że w przypadku zbiorów metoda `[]` działa dokładnie tak samo jak w przypadku tablic mieszających. Metoda `new` może otrzymywać na wejściu opcjonalny obiekt wyliczeniowy i opcjonalny blok kodu. Jeśli zdecydujemy się na użycie bloku, przekazany kod zostanie wykorzystany w roli swoistego preprocesora przetwarzającego daną listę (podobnie jak operacja `map`):

```
s1 = Set[3,4,5]           # czyli {3,4,5} w notacji matematycznej
arr = [3,4,5]
s2 = Set.new(arr)        # jw.
s3 = Set.new(arr) { |x| x.to_s } # zbiór łańcuchów (nie liczb)
```

9.1.1. PROSTE OPERACJE NA ZBIORACH

Do wyznaczania sumy zbiorów służy metoda `union` (dla metody `union` zdefiniowano dwa aliasy: `|` oraz `+`):

```
x = Set[1,2,3]
y = Set[3,4,5]
```

```

a = x.union(y)    #Set[1,2,3,4,5]
b = x | y         #jw.
c = x + y        #jw.

```

Iloczyn (część wspólną) dwóch zbiorów możemy wyznaczać za pomocą metody `intersection` (lub jej aliasu `&`):

```

x = Set[1,2,3]
y = Set[3,4,5]

a = x.intersection(y)    #Set[3]
b = x & y                 #jw.

```

Dwuargumentowy operator – reprezentuje operację różnicy zbiorów (o czym wspominaliśmy już przy okazji omawiania tablic w punkcie 8.1.9 zatytułowanym „Stosowanie tablic w roli zbiorów matematycznych”):

```
diff = Set[1,2,3] - Set[3,4,5]    #Set[1,2]
```

Podobnie jak w przypadku tablic, przynależność do zbioru możemy sprawdzać za pomocą metod `member?` i `include?`. Warto pamiętać, że kolejność operandów jest odwrotna niż ta, którą znamy z lekcji matematyki:

```

Set[1,2,3].include?(2)    #true
Set[1,2,3].include?(4)    #false
Set[1,2,3].member?(4)     #false

```

Za pomocą metody `empty?` możemy (podobnie jak w przypadku tabel) sprawdzić, czy mamy do czynienia ze zbiorem pustym. Metoda `clear` usuwa wszystkie elementy zbioru niezależnie od jego bieżącej zawartości:

```

s = Set[1,2,3,4,5,6]
s.empty?    #false
s.clear
s.empty?    #true

```

Klasa `Set` oferuje też możliwość określania relacji łączących dwa zbiory — za pomocą odpowiednich metod możemy sprawdzać, czy zbiór reprezentowany przez obiekt docelowy jest podzbiorem innego zbioru, czy jest jego podzbiorem właściwym lub czy jest jego nadzbiorem:

```

x = Set[3,4,5]
y = Set[3,4]

x.subset?(y)    #false
y.subset?(x)    #true
y.proper_subset?(x)    #true
x.subset?(x)    #true
x.proper_subset?(x)    #false
x.superset?(y)    #true

```

Metoda `add` (alias `<<`) dodaje pojedynczy element do wskazanego zbioru i zwraca zmodyfikowany zbiór; metoda `add?` zwraca wartość `nil`, jeśli wskazany zbiór zawiera już dany element. Za pomocą metody `merge` możemy jednocześnie dodawać wiele elementów. Wszystkie wymienione metody mogą oczywiście modyfikować obiekt docelowy (reprezentujący zbiór). Metoda `replace` działa na zbiorach dokładnie tak jak w przypadku łańcuchów i tablic.

I wreszcie operator `==` służy do weryfikacji równości dwóch zbiorów:

```
Set[3,4,5] == Set[5,4,3]    # true
```

9.1.2. ZAAWANSOWANE OPERACJE NA ZBIORACH

Oczywiście mamy też możliwość iteracyjnego przeszukiwania elementów zbioru, jednak (podobnie jak w przypadku tablic mieszających) nie należy oczekiwać określonego porządku, ponieważ zbiory z natury rzeczy są strukturami nieuporządkowanymi, a język Ruby nie gwarantuje powtarzalności sekwencji ich elementów. (Mimo że w kolejnych operacjach iteracyjnego przeszukiwania możemy otrzymywać elementy zbioru w różnej kolejności, wykorzystywanie tego faktu do budowy wiarygodnego generatora sekwencji losowych byłoby naiwnością).

```
s = Set[1,2,3,4,5]
s.each {|x| puts x; break }    # Dane wynikowe: 5
```

Metoda `classify` jest odpowiednikiem metody `partition` (właśnie metoda `classify` była naszą inspiracją podczas prac nad identycznie nazwaną funkcją w punkcie 8.3.3 zatytułowanym „Metoda `partition`”):

```
files = Set.new(Dir["*"])
hash = files.classify do |f|
  if File.size(f) <= 10_000
    :small
  elsif File.size(f) <= 10_000_000
    :medium
  else
    :large
  end
end
```

```
big_files = hash[:large]    # Zmienna big_files jest obiektem klasy Set.
```

Podobnie działa metoda `divide`, która do określania przynależności zbiorów do poszczególnych grup wykorzystuje przekazany blok kodu, po czym zwraca wynik w postaci zbioru zbiorów.

Jeśli blok wejściowy obejmuje tylko jeden operand, wywołania wykonywane przez metodę `divide` będą miały następującą postać: `block.call(a) == block.call(b)`. W ten sposób metoda `divide` określa, czy elementy `a` i `b` należą do tego samego podzbioru. Jeśli prześlemy blok obejmujący dwa operandy, metoda `divide` będzie

wykonywała wywołania `block.call(a,b)` celem określenia, czy te dwa elementy należą do tego samego zbioru.

Przykładowo, poniższy blok kodu (obejmujący jeden operand) dzieli zbiór wejściowy na dwa zbiory wynikowe, z których jeden zawiera liczby parzyste, drugi zawiera liczby nieparzyste:

```
require 'set'
numbers = Set[1,2,3,4,5,6,7,8,9,0]
set = numbers.divide{|i| i % 2}
p set # =<Set: {#<Set: {5, 1, 7, 3, 9}>, #<Set: {0, 6, 2, 8, 4}>}>
```

Poniżej przedstawiono jeszcze jeden ciekawy przykład. Liczby bliźniacze to takie liczby pierwsze, których różnica wynosi 2 (czyli np. 11 i 13); pozostałe liczby pierwsze (np. 23) muszą się różnić od najbliższych liczb pierwszych przynajmniej o 3. Poniższy fragment kodu dokonuje podziału na te dwie grupy i umieszcza sekwencje bliźniaczych liczb pierwszych w odrębnych podzbiorach. W prezentowanym przykładzie wykorzystano blok obejmujący dwa operandy:

```
primes = Set[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
set = primes.divide{|i,j| (i-j).abs == 2}
# Zmienna set zawiera teraz zbiór: #<Set: {#<Set: {23}>, #<Set: {11, 13}>,
#                               #<Set: {17, 19}>, #<Set: {5, 7, 3}>,
#                               #<Set: {2}>, #<Set: {29, 31}>}>
# Krótszy zapis: {{23},{11,13},{17,19},{5,7,3},{2},{29,31}}
```

Prezentowana metoda jest trudna w interpretacji, a jej wywołania mogą być błędnie interpretowane. Sam wolę stosować metodę `classify`, która jest dużo bardziej intuicyjna.

Warto pamiętać, że klasa `Set` nie we wszystkich przypadkach nakłada na nas obojętne stosowania wyłącznie parametrów lub operandów reprezentujących zbiory. (Czytelnicy, którym analizowane mechanizmy wydają się niejasne, powinni wrócić do omówienia techniki *duck typing* w rozdziale 1.). Praktyka pokazuje, że większość spośród prezentowanych metod może otrzymywać na wejściu (w formie operandów) **dowolne obiekty reprezentujące struktury wyliczeniowe**. Taka możliwość niewątpliwie zwiększa elastyczność omawianych rozwiązań.

Na zbiorach można wykonywać też inne przydatne operacje (w tym wszystkie metody modułu `Enumerable`). Ponieważ nie zdecydowaliśmy się na analizę takich operacji jak `flatten`, Czytelnicy zainteresowani szczegółowym omówieniem tej i innych metod powinni szukać odpowiednich materiałów na witrynie <http://ruby-doc.org/>.

9.2. PRACA ZE STOSAMI I KOLEJKAMI

Stosy i kolejki to dwie pierwsze struktury danych (spośród tych, którymi się zajmujemy), których obsługa nie została zaimplementowana w formie mechanizmów wbudowanych języka Ruby. Oznacza to, że programiści tego języka nie mają do dyspozycji klas

Stack ani Queue podobnych do klas Array czy Hash (z wyjątkiem klasy Queue biblioteki *thread.rb*, którą omówimy w dalszej części tego podrozdziału).

Mimo to język Ruby oferuje swoim programistom pewne rozwiązania umożliwiające implementację struktur stosu i kolejek. Okazuje się, że klasa Array implementuje całą funkcjonalność niezbędną do przetwarzania tablic w sposób właściwy dla stosów i kolejek. Odpowiednie rozwiązania przeanalizujemy w dalszej części tego podrozdziału.

Stos (ang. *stack*) jest strukturą LIFO (od ang. *last-in first-out*; ostatni na wejściu, pierwszy na wyjściu). Tradycyjnym przykładem ilustrującym działanie stosu są talerze układane jeden na drugim — talerz położony na stosie jako ostatni jest też jako pierwszy zdejmowany z tego stosu.

Zbiór operacji, które można wykonywać na zbiorach, jest dość ograniczony. Do podstawowych działań należy **kładzenie elementów na stosie** (ang. *push*) i **zdejmowanie elementów ze stosu** (ang. *pop*). Wiele implementacji dodatkowo oferuje operacje umożliwiające wykrywanie stosu pustego oraz uzyskiwanie dostępu do elementu na szczycie stosu bez jego zdejmowania (usuwanie z przetwarzanej struktury). Żadna implementacja stosu nie oferuje mechanizmów, za pośrednictwem których moglibyśmy uzyskiwać dostęp do elementów składowanych poniżej szczytu stosu (w jego środku).

Wielu Czytelników zapewne zadaje sobie pytanie: „Jak tablica może implementować stos, skoro dostęp do każdego z jej elementów można uzyskiwać w dowolnym momencie?”. Odpowiedź jest bardzo prosta — stos jest implementowany na wyższym poziomie abstrakcji niż tablica i jako taki pozostaje stosem, dopóki jest odpowiednio traktowany przez programistę. Oznacza to, że w chwili uzyskania dostępu do elementu składowanego poniżej szczytu stosu, przetwarzana struktura faktycznie przestaje być stosem.

Oczywiście możemy bez trudu zdefiniować klasę Stack reprezentującą strukturę danych, której elementy będą udostępniane wyłącznie według reguł obowiązujących w stosach. W dalszej części tego podrozdziału przeanalizujemy sposób realizacji tego zadania.

Warto pamiętać, że wiele algorytmów operujących na stosach wykorzystuje dobrze przemyślane, eleganckie mechanizmy rekurencyjne. Wystarczy poświęcić dosłownie chwilę, by zrozumieć, dlaczego właśnie stos wprost idealnie nadaje się do tego rodzaju zastosowań. Każde kolejne wywołanie funkcji lub metody powoduje umieszczenie nowych danych na stosie systemowym, które są z tego stosu zdejmowane po zwróceniu sterowania do funkcji lub metody wywołującej. Oznacza to, że algorytm rekurencyjny faktycznie zastępuje stos zdefiniowany wprost przez użytkownika niejawnym (ukrytym) stosem systemowym. Który z tych stosów jest lepszy? Wszystko zależy od wymaganej dostępności danych, efektywności i kilku innych czynników.

Kolejka (ang. *queue*) jest strukturą FIFO (od ang. *first-in first-out*; pierwszy na wejściu, pierwszy na wyjściu). Bodaj najprostszą analogią, jaką można wskazać w codziennym życiu, jest przykład kolejki do kasy kinowej. Nowi widzowie, którzy dopiero weszli do kina, z natury rzeczy ustawiają się na końcu kolejki, a kasa obsługuje w pierwszej kolejności tych widzów, którzy oczekują w kolejce najdłużej. W większości obszarów programowania kolejki są wykorzystywane rzadziej niż stosy.

Kolejki okazują się szczególnie przydatne w środowiskach czasu rzeczywistego, gdzie elementy danych są przetwarzane natychmiast po ich dostarczeniu do systemu. Z kolejek często korzystają programiści implementujący mechanizmy typu producent-konsument (szczególnie w środowiskach wielowątkowych lub wielozadaniowych). Dobrym przykładem takiego zastosowania tej struktury jest kolejka zadań drukowania — kolejne zadania są dodawane na koniec kolejki i oczekują na realizację do chwili wykonania wcześniejszych zadań.

Do dwóch tradycyjnie (zgodnie z literaturą) najważniejszych operacji na kolejkach należy **ustawianie w kolejce** (ang. *enqueue*) i **odłączanie od kolejki** (ang. *dequeue*). Metody egzemplarzy zdefiniowane w klasie `Array`, które odpowiadają za realizację tych zadań, nazwano odpowiednio `unpush` i `shift`.

Warto pamiętać, że inna metoda klasy `Array`, nazwana `unshift`, może być wykorzystywana wraz ze wspomnianą metodą `shift` do implementacji stosu (nie kolejki), ponieważ metoda `unshift` dodaje element na tym samym końcu struktury danych, z którego metoda `shift` usuwa element. Stosując rozmaite kombinacje tych metod, możemy implementować zarówno stosy, jak i kolejki, jednak nie będziemy analizowali wszystkich możliwych rozwiązań.

Na tym możemy zakończyć nasze wprowadzenie w świat stosów i kolejek. W kolejnych punktach tego podrzdziału przeanalizujemy kilka praktycznych przykładów.

9.2.1. IMPLEMENTACJA STOSU WYMUSZAJĄCEGO WŁAŚCIWY DOSTĘP DO DANYCH

Wspominaliśmy już, że zajmiemy się problemem implementacji stosu w sposób uniemożliwiający uzyskiwanie dostępu do niewłaściwych elementów (spoza szczytu tej struktury). Odpowiednią klasę zaimplementujemy w tym punkcie. Poniżej przedstawiono prostą klasę wykorzystującą tablicę wewnętrzną i zarządzającą dostępem do jej elementów. (Prezentowane zadanie można zrealizować także w inny sposób — stosując np. technikę delegacji — jednak poniższe rozwiązanie jest dużo prostsze i zdaje egzamin w praktyce).

```
class Stack

  def initialize
    @store = []
  end
```



```

def push(x)
  @store.push x
end

def pop
  @store.pop
end

def peek
  @store.last
end

def empty?
  @store.empty?
end

end

```

W klasie `Stack` zaimplementowaliśmy jedną dodatkową operację, której nie zdefiniowano dla tablic. Działanie metody `peek` sprowadza się do odczytania i zwrócenia elementu ze szczytu stosu bez jego usuwania z tej struktury.

W niektórych spośród przykładów prezentowanych w dalszej części tego podrozdziału będziemy zakładali, że klasa `Stack` została zdefiniowana.

9.2.2. WYKRYWANIE NIEZBILANSOWANYCH ZNAKÓW INTERPUNKCYJNYCH W WYRAŻENIACH

Poprawność wyrażeń obejmujących grupy podwyrażeń (otaczane nawiasami okrągłymi, klamrowymi i ostrymi) można sprawdzać z wykorzystaniem stosu. Dla każdego kolejnego poziomu zagnieżdżenia wyrażeń rozmiar stosu jest zwiększany o jeden element (poziom) — kiedy odnajdujemy odpowiedni symbol zamykający, możemy zdjąć ze stosu odpowiedni symbol otwierający. Jeśli wykryjemy symbol (nawias) zamykający, który nie pasuje do symbolu otwierającego reprezentowanego na szczycie stosu, lub jeśli po przetworzeniu całego tekstu okaże się, że stos nie jest pusty, możemy być pewni, że format badanego wyrażenia jest nieprawidłowy.

```

def paren_match(str)
  stack = Stack.new
  lsym = "{[(<"
  rsym = "}]>"
  str.each_byte do |byte|
    sym = byte.chr
    if lsym.include? sym
      stack.push(sym)
    elsif rsym.include? sym
      top = stack.peek
      if lsym.index(top) != rsym.index(sym)
        return false
      end
    end
  end
end

```

```

else
  stack.pop
end
# Ignorujemy znaki, które nie należą do żadnej grupy...
end
end
# Upewniamy się, że dany stos jest pusty...
return stack.empty?
end

str1 = "(((a+b))*((c-d)-(e*f)))"
str2 = "[[(a-(b-c))], [[x,y]]]"

paren_match str1      # false
paren_match str2      # true

```

Zagnieżdżony charakter wyrażeń powoduje, że problem weryfikacji ich poprawności wprost idealnie nadaje się do rozwiązania z użyciem stosu. Bardziej skomplikowanym problemem byłoby wykrywanie niezbilansowanych znaczników w kodzie HTML i XML. W takim przypadku pojedyncze tokeny składałyby się z wielu znaków (nie — jak w powyższym przykładzie — z samych nawiasów), jednak struktura i logika samego rozwiązania pozostałyby identyczne. Innym ciekawym przykładem problemu, którego rozwiązanie z reguły wymaga użycia stosu, jest konwersja wyrażeń w notacji infiksowej (wzrostkowej) na wyrażenia w notacji postfiksowej (przyrostkowej) i odwrotnie, przetwarzanie wyrażeń w notacji postfiksowej (realizowane przez wiele interpreterów, w tym wirtualną maszynę Javy) oraz niemal wszystkie problemy rozwiązywane za pomocą algorytmów rekurencyjnych. W kolejnym punkcie dokonamy krótkiej analizy relacji łączącej stosy z mechanizmami rekurencyjnymi.

9.2.3. STOSY I REKURENCJA

Nasze rozważania poświęcone relacji izomorfizmu łączącej algorytmy wykorzystujące stos z algorytmami rekurencyjnymi zilustrujemy przykładem klasycznego problemu wież Hanoi.

Legenda głosi, że gdzieś na Dalekim Wschodzie istnieje starożytna świątynia, w której mnisi w skupieniu próbują rozwiązać problem przenoszenia dysków pomiędzy trzema palami według określonych reguł ograniczających wykonywane ruchy. Pierwszy stos (wieża) składa się z 64 dysków — kiedy wszystkie te dyski uda się przenieść na trzeci stos, zgodnie z przytoczoną legendą nastąpi koniec świata.

Wielu ludzi stara się za wszelką cenę rozwiązywać podobne problemy i wyjaśniać związane z nimi tajemnice. Wygląda na to, że w rzeczywistości opisywany problem został wymyślony dopiero w roku 1883 przez francuskiego matematyka Edouarda Lucasa, który nigdy nie zajmował się kulturą Dalekiego Wschodu. Co więcej, sam Lucas nazwał tę zagadkę problemem „wieży Hanoi” (w liczbie pojedynczej).

Jeśli więc pierwsze zdania tego punktu wprawiły Czytelnika w przerażenie, uspoko-
jam — mityczni mnisi najprawdopodobniej nie doprowadzą do końca świata. Warto
przy tej okazji wspomnieć, że przekładanie 64 dysków wymagałoby od nich wykonania
 $2^{64}-1$ ruchów. Wystarczy kilka minut z dobrym kalkulatorem, by stwierdzić, że roz-
wiązanie tego problemu zajęłoby mitycznym mnichom kilka milionów lat.

Wróćmy teraz do reguł naszej gry. (Przypomnimy je, mimo że niemal wszyscy stu-
denci pierwszego roku informatyki na całym świecie powinni je doskonale znać). Na
początku dysponujemy palem, na który ułożono stos złożony z określonej liczby dys-
ków — od tej chwili będziemy go nazywać **palem źródłowym**. Naszym zadaniem jest
przeniesienie wszystkich tych dysków na **pal docelowy** z wykorzystaniem trzeciego
pala zewnętrznego, który służy nam wyłącznie do tymczasowego przechowywania
dysków. Problem w tym, że dyski możemy przenosić pojedynczo i nigdy większy
dysk nie może się znaleźć ponad mniejszym.

W poniższym fragmencie kodu źródłowego rozwiązaliśmy ten problem, wykorzystując
strukturę stosu. Zdecydowaliśmy się przenieść tylko 3 dyski, ponieważ przenoszenie
64 dysków zajęłoby przeciętnemu komputerowi mnóstwo czasu:

```
def towers(list)
  while !list.empty?
    n, src, dst, aux = list.pop
    if n == 1
      puts "Przenosimy dysk z pala #{src} na pal #{dst}"
    else
      list.push [n-1, aux, dst, src]
      list.push [1, src, dst, aux]
      list.push [n-1, src, aux, dst]
    end
  end
end

list = []
list.push([3, "a", "c", "b"])

towers(list)
```

Poniżej przedstawiono dane wyjściowe wyświetlone przez nasz algorytm:

```
Przenosimy dysk z pala a na pal c
Przenosimy dysk z pala a na pal b
Przenosimy dysk z pala c na pal b
Przenosimy dysk z pala a na pal c
Przenosimy dysk z pala b na pal a
Przenosimy dysk z pala b na pal c
Przenosimy dysk z pala a na pal c
```

Klasycznym rozwiązaniem tego problemu jest oczywiście algorytm rekurencyjny. Jak już wspominaliśmy, bliska relacja łącząca oba algorytmy jest o tyle naturalna, że każdy algorytm rekurencyjny i tak wymaga niejawnego składowania wyników pośrednich na stosie systemowym:

```
def towers(n, src, dst, aux)
  if n==1
    puts "Przenosimy dysk z pala #{src} na pal #{dst}"
  else
    towers(n-1, src, aux, dst)
    towers(1, src, dst, aux)
    towers(n-1, aux, dst, src)
  end
end

towers(3, "a", "c", "b")
```

Okazuje się, że przedstawiony powyżej algorytm rekurencyjny wyświetli dokładnie te same dane wynikowe. Co ciekawe, nie mogliśmy się powstrzymać i porównaliśmy czasy wykonywania obu metod — niech to będzie naszą tajemnicą, że wersja rekurencyjna okazała się blisko dwukrotnie szybsza.

9.2.4. IMPLEMENTACJA KOLEJKI WYMUSZAJĄCEJ WŁAŚCIWY DOSTĘP DO DANYCH

W niniejszym punkcie zdefiniujemy wyspecjalizowaną klasę kolejki (nazwaną Queue) w sposób bardzo przypominający ten, który zastosowaliśmy w przypadku klasy Stack. Jeśli chcemy się zabezpieczyć przed niewłaściwym dostępem do elementów naszej struktury danych, koniecznie powinniśmy zdefiniować podobną klasę:

```
class Queue

  def initialize
    @store = []
  end

  def enqueue(x)
    @store << x
  end

  def dequeue
    @store.shift
  end

  def peek
    @store.first
  end
end
```

```

def length
  @store.length
end

def empty?
  @store.empty?
end

end

```

Warto przy tej okazji wspomnieć, że biblioteka *thread.rb* zawiera klasę *Queue*, która doskonale się sprawdza w środowisku wielowątkowym. Co więcej, istnieje też nieco zmodyfikowana odmiana tej metody nazwana *SizedQueue*.

Wspomniane klasy definiują metody nazwane *enq* i *deq*, czyli odpowiedniki naszych metod *enqueue* i *dequeue*. Co ciekawe, metody *Queue* i *SizedQueue* biblioteki *thread.rb* definiują dla tych operacji aliasy *push* i *pop*, co jest o tyle niezrozumiałe, że w przeciwieństwie do stosu kolejka jest strukturą danych FIFO (nie LIFO).

Klasa *Queue* biblioteki *thread.rb* oczywiście gwarantuje bezpieczeństwo przetwarzania wielowątkowego, co dla wielu programistów może być istotne. Jeśli okaże się, że potrzebujemy klasy *Stack* oferującej podobne możliwości, w pierwszej kolejności powinniśmy zmodyfikować niezbędne mechanizmy istniejącej klasy *Queue* (taka zmiana nie powinna nam zająć zbyt wiele czasu i raczej nie sprawi nam kłopotu).

W pierwszym wydaniu tej książki przedstawiono dość długi przykład praktycznego wykorzystania struktury danych kolejki. Podobnie jak w przypadku części przykładów ilustrujących zastosowania stosów w tym wydaniu zrezygnowaliśmy z jego prezentacji z uwagi na brak miejsca.

9.3. PRACA Z DRZEWAMI

*Wątpię, bym kiedy mógł opiewać
wiersz, co dorówna pięknu drzewa.*

— *Trees, [Alfred] Joyce Kilmer*

W świecie komputerów drzewa z pewnością należą do grupy stosunkowo intuicyjnych struktur danych (mimo że z reguły rysuje się je w taki sposób, że korzeń znajduje się na samej górze, a liście na dole). Wynika to z faktu, że w życiu codziennym mamy do czynienia z bardzo wieloma rodzajami danych hierarchicznych. Do najbardziej popularnych struktur tego typu należą drzewa genealogiczne, schematy organizacyjne przedsiębiorstw oraz struktury katalogów na dyskach twardych.

Terminologia związana z drzewami jest dość bogata, ale jej zrozumienie nie powinno nam sprawić najmniejszych problemów. Elementy drzew nazywamy **węzłami** (ang.

nodes); pierwszy, najwyższy węzeł nazywamy **korzeniem** (ang. *root*). Węzeł może mieć **potomków** (ang. *descendants*); o bezpośrednich potomkach mówimy, że są **dziećmi** (ang. *children*) danego węzła. Podobnie węzeł może mieć jednego **rodzica** (ang. *parent*) i wielu pośrednich **przodków** (ang. *ancestors*). Węzeł, dla którego nie istnieją węzły potomne, nazywamy **liściem** (ang. *leaf*). Poddrewo drzewa składa się z wybranego węzła i wszystkich jego węzłów potomnych. Przeszukiwanie drzewa bywa określane mianem jego **trawersowania** (ang. *traversing*).

W dalszej części tego podrozdziału skoncentrujemy się przede wszystkim na drzewach binarnych, chociaż w praktyce dla pojedynczego węzła może istnieć dowolna liczba bezpośrednich potomków. Przeanalizujemy sposoby tworzenia, wypełniania i przeszukiwania drzew, po czym przyjrzymy się kilku praktycznym zadaniom, których realizacja jest prostsza dzięki tej ciekawej strukturze danych.

Czytelnicy dowiedzą się, że w wielu językach programowania (w tym w języku C i w Pascalu) drzewa są implementowane za pomocą prawdziwych wskaźników do adresów pamięciowych. Z drugiej strony, w wielu współczesnych językach (w tym w języku Ruby i w Javie) w ogóle nie musimy stosować wskaźników — referencje do obiektów sprawdzają się równie dobrze lub nawet lepiej.

9.3.1. IMPLEMENTACJA DRZEWA BINARNEGO

Strukturę drzewa binarnego można zaimplementować w języku Ruby na wiele sposobów. Możemy na przykład użyć standardowej tablicy do składowania elementów drzewa. W niniejszym punkcie zastosujemy bardziej tradycyjne podejście, które pod wieloma względami przypomina rozwiązanie stosowane przez programistów języka C (z tą różnicą, że wskaźniki zastąpimy referencjami do obiektów).

Czego potrzebujemy do jednoznacznego opisywania drzewa binarnego? Każdy węzeł wymaga oczywiście atrybutu reprezentującego składowane w nim dane. Każdy węzeł dodatkowo musi obejmować parę atrybutów wskazujących na lewe i prawe poddrewo znajdujące się poniżej tego węzła.

Musimy też znaleźć sposób wstawiania nowych elementów do drzewa binarnego i efektywnego uzyskiwania dostępu do informacji reprezentowanych przez to drzewo. Realizacja tych zadań będzie wymagała implementacji dwóch wyspecjalizowanych metod.

Pierwsze drzewo, które przeanalizujemy, będzie implementowało te dwie metody w dość niekonwencjonalny sposób. Klasę `Tree` będziemy następnie rozwijali w kolejnych przykładach (prezentowanych w dalszych punktach tego podrozdziału).

Drzewo w pewnym sensie jest definiowane przez algorytm wstawiania nowych elementów i sposób przeszukiwania istniejących węzłów. W naszym pierwszym przykładzie (przedstawionym w listingu 9.1) zdefiniowano metodę `insert` wstawiającą elementy techniką **wszecz** (ang. *breadth-first*), czyli od góry do dołu i od lewej do

prawej strony. Takie rozwiązanie gwarantuje nam możliwie niewielki wzrost głębokości drzewa i utrzymywanie tej struktury w stanie zrównoważonym. Klasę `Tree` uzupełniono o metodę `traverse`, która przeszukuje iteracyjnie węzły drzewa (także wszerz).

LISTING 9.1. Wstawianie elementów i przeszukiwanie węzłów drzewa wszerz

```
class Tree

  attr_accessor :left
  attr_accessor :right
  attr_accessor :data

  def initialize(x=nil)
    @left = nil
    @right = nil
    @data = x
  end

  def insert(x)
    list = []
    if @data == nil
      @data = x
    elsif @left == nil
      @left = Tree.new(x)
    elsif @right == nil
      @right = Tree.new(x)
    else
      list << @left
      list << @right
      loop do
        node = list.shift
        if node.left == nil
          node.insert(x)
          break
        else
          list << node.left
        end
        if node.right == nil
          node.insert(x)
          break
        else
          list << node.right
        end
      end
    end
  end

  def traverse()
    list = []
  end
end
```

```

yield @data
list << @left if @left != nil
list << @right if @right != nil
loop do
  break if list.empty?
  node = list.shift
  yield node.data
  list << node.left if node.left != nil
  list << node.right if node.right != nil
end
end

end

items = [1, 2, 3, 4, 5, 6, 7]

tree = Tree.new

items.each {|x| tree.insert(x)}

tree.traverse {|x| print "#{x} "}
print "\n"

# Wyświetli "1 2 3 4 5 6 7".

```

Prezentowany rodzaj drzewa (definiowany przez algorytmy wstawiania i przeszukiwania węzłów) nie jest szczególnie interesujący. Przedstawiona klasa `Tree` może jednak stanowić cenne wprowadzenie w świat bardziej zaawansowanych struktur drzewiastych.

9.3.2. SORTOWANIE DANYCH Z WYKORZYSTANIEM DRZEWA BINARNEGO

Drzewo binarne doskonale nadaje się do sortowania przypadkowych danych. (W przypadku danych już posortowanych drzewo binarne nie różni się od listy jednokierunkowej). Źródłem niezwykłej efektywności drzewa binarnego jest fakt, że każde porównanie eliminuje połowę pozostałych możliwości w zakresie wyboru właściwego miejsca dla nowego węzła.

Mimo że znajomość tego rozwiązania wśród dzisiejszych informatyków jest dość powszechna, zdecydowałem się przedstawić implementację tego algorytmu napisaną w języku Ruby. Kod zawarty w listingu 9.2 zbudowano na bazie poprzedniego przykładu.

LISTING 9.2. Sortowanie danych z wykorzystaniem drzewa binarnego

```

class Tree

  # Zakładamy, że dysponujemy definicją
  # z poprzedniego przykładu...

  def insert(x)
    if @data == nil
      @data = x
    elsif x <= @data
      if @left == nil
        @left = Tree.new x
      else
        @left.insert x
      end
    else
      if @right == nil
        @right = Tree.new x
      else
        @right.insert x
      end
    end
  end

  def inorder()
    @left.inorder {|y| yield y} if @left != nil
    yield @data
    @right.inorder {|y| yield y} if @right != nil
  end

  def preorder()
    yield @data
    @left.preorder {|y| yield y} if @left != nil
    @right.preorder {|y| yield y} if @right != nil
  end

  def postorder()
    @left.postorder {|y| yield y} if @left != nil
    @right.postorder {|y| yield y} if @right != nil
    yield @data
  end
end

items = [50, 20, 80, 10, 30, 70, 90, 5, 14,
         28, 41, 66, 75, 88, 96]

tree = Tree.new

items.each {|x| tree.insert(x)}

```

```

tree.inorder {|x| print x, " "}
print "\n"
tree.preorder {|x| print x, " "}
print "\n"
tree.postorder {|x| print x, " "}
print "\n"

# Dane wyjściowe:
# 5 10 14 20 28 30 41 50 66 70 75 80 88 90 96
# 50 20 10 5 14 30 28 41 80 70 66 75 90 88 96
# 5 14 10 28 41 30 20 66 75 70 88 96 90 80 50 print "\n"

```

9.3.3. STOSOWANIE DRZEWA BINARNEGO W ROLI TABLICY WYSZUKIWANIA

Przypuśćmy, że dysponujemy posortowanym drzewem. Posortowane struktury drzewiaste tradycyjnie sprawdzają się w roli tablic wyszukiwania; przykładowo, odnalezienie określonego węzła w posortowanym drzewie zrównoważonym reprezentującym milion elementów wymaga (w najgorszym przypadku) zaledwie dwudziestu operacji porównania (głębokość takiego drzewa jest równa logarytmowi o podstawie równej 2 z liczby węzłów). Aby prezentowane rozwiązanie w jak największym stopniu ilustrowało rzeczywiste zastosowania, zakładamy, że dane składowane w poszczególnych węzłach nie mają postaci pojedynczych wartości, tylko kluczy wskazujących na powiązane ze sobą dane.

W większości (jeśli nie we wszystkich sytuacjach) tablice mieszające lub tabele zewnętrznych baz danych lepiej sprawdzają się w tej roli. Mimo to warto przeanalizować przedstawiony poniżej przykład kodu źródłowego:

```

class Tree

  # Zakładamy, że dysponujemy definicjami
  # z wcześniejszych przykładów...

  def search(x)
    if self.data == x
      return self
    elsif x < self.data
      return left ? left.search(x) : nil
    else
      return right ? right.search(x) : nil
    end
  end

end

keys = [50, 20, 80, 10, 30, 70, 90, 5, 14,
        28, 41, 66, 75, 88, 96]

```

```

tree = Tree.new

keys.each {|x| tree.insert(x)}

s1 = tree.search(75)  # Zwraca referencję do węzła zawierającego
                    # wartość 75...

s2 = tree.search(100) # Zwraca wartość nil (żądanego węzła nie odnaleziono).

```

9.3.4. KONWERSJA DRZEWA NA ŁAŃCUCH LUB TABLICĘ

Te same zabiegi, które umożliwiają efektywne przeszukiwanie drzewa, możemy z powodzeniem wykorzystywać do konwertowania tego rodzaju struktur na łańcuchy i tablice. Tym razem przeszukujemy drzewo w porządku **inorder**, chociaż równie dobrze moglibyśmy zastosować inną technikę:

```

class Tree

  # Zakładamy, że dysponujemy definicjami
  # z wcześniejszych przykładów...

  def to_s
    "[" +
      if left then left.to_s + "," else "" end +
      data.inspect +
      if right then "," + right.to_s else "" end + "]"
  end

  def to_a
    temp = []
    temp += left.to_a if left
    temp << data
    temp += right.to_a if right
    temp
  end

end

items = %w[bongo grimace monoid jewel plover nexus synergy]

tree = Tree.new
items.each {|x| tree.insert x}

str = tree.to_a * ","
# Zmienna str zawiera teraz łańcuch "bongo,grimace,jewel,monoid,nexus,plover,synergy".
arr = tree.to_a
# Zmienna arr zawiera teraz tablicę:
# ["bongo",["grimace",[["jewel"],["monoid",[["nexus"],["plover",
# ["synergy"]]]]]]]

```

Warto zwrócić uwagę na fakt, że tablica wynikowa jest zagnieżdżona w stopniu odpowiadającym głębokości drzewa, na podstawie którego została wygenerowana. Możemy oczywiście użyć metody `flatten` do przekształcenia tej tablicy w strukturę niezagnieżdżoną (płaską).

9.4. PRACA Z GRAFAMI

Graf jest kolekcją węzłów, które mogą być ze sobą połączone w dowolny sposób. (Specyficznym przykładem grafu jest drzewo). Nie będziemy się zagłębiać w szczegółowe analizy zagadnień związanych z grafami, ponieważ przedstawienie niezbędnej teorii i terminologii zajęłoby nam mnóstwo czasu. Zanim jednak przystąpimy do prezentacji konkretnych przykładów, poświęcimy chwilę na ogólne rozważania na temat szeroko rozumianej informatyki i pewnych obszarów ze świata matematyki.

Okazuje się, że grafy mają wiele praktycznych zastosowań. Wyobraźmy sobie na przykład tradycyjny atlas samochodowy z drogami łączącymi miasta i miejscowości lub diagram obwodów elektrycznych. Najlepszą formą reprezentowania obu tych struktur jest właśnie graf. Teoria grafów doskonale nadaje się także do reprezentowania sieci komputerowych, niezależnie od tego, czy są to proste sieci LAN, duże struktury obejmujące wiele systemów, czy cały internet złożony z milionów węzłów.

Mówiąc o grafach, zwykle mamy na myśli **grafy nieskierowane** (ang. *undirected graphs*). Najprościej mówiąc, graf nieskierowany to taki, w którym linie łączące poszczególne węzły nie są zakończone grotami (nie mają postaci strzałek); a dwa węzły mogą być albo połączone, albo nie. Inną formą grafu jest **graf skierowany** (ang. *directed graph, digraph*), który może zawierać „ulice jednokierunkowe”. W grafie skierowanym fakt połączenia węzła x z węzłem y nie oznacza, że istnieje połączenie w przeciwnym kierunku. Węzły bywają określane mianem **wierzchołków** (ang. *vertexes*). I wreszcie istnieje pojęcie **grafu ważonego** (ang. *weighted graph*), w którym poszczególne połączenia (krawędzie) mają przypisane wagi reprezentujące np. „odległości” dzielące połączone węzły. Na tym możemy zakończyć nasze wprowadzenie w świat grafów — Czytelnicy zainteresowani pogłębieniem swojej wiedzy mogą skorzystać z niezliczonych publikacji zarówno z dziedziny informatyki, jak i matematyki.

W języku Ruby (podobnie jak w większości języków programowania) graf może być reprezentowany na wiele różnych sposobów — w tym w formie prawdziwej sieci wzajemnie powiązanych obiektów lub macierzy zawierającej dane o krawędziach danego grafu. W dalszej części tego podrozdziału przeanalizujemy obie te reprezentacje i zapoznamy się z kilkoma praktycznymi przykładami przetwarzania grafów.

9.4.1. IMPLEMENTACJA GRAFU W FORMIE MACIERZY SĄSIEDZTWA

W niniejszym punkcie przeanalizujemy przykład zbudowany na bazie dwóch wcześniejszych przykładów. Kod zawarty w listingu 9.3 implementuje reprezentację grafu nieskierowanego w formie tzw. macierzy sąsiedztwa (ang. *adjacent matrix*). W prezentowanej implementacji wykorzystano klasę `ZArray` (punkt 8.1.26 zatytułowany „Określanie wartości domyślnej dla nowych elementów tablicy”) celem zagwarantowania, że wszystkie nowe elementy będą zawierały wartość zerową, oraz zastosowano technikę dziedziczenia po klasie `TriMatrix` (punkt 8.1.7 zatytułowany „Stosowanie wyspecjalizowanych funkcji indeksujących”), aby reprezentować grafy w formie **mniejszych macierzy trójkątnych**.

LISTING 9.3. Implementacja macierzy sąsiedztwa

```
class LowerMatrix < TriMatrix

  def initialize
    @store = ZArray.new
  end

end

class Graph

  def initialize(*edges)
    @store = LowerMatrix.new
    @max = 0
    for e in edges
      e[0], e[1] = e[1], e[0] if e[1] > e[0]
      @store[e[0],e[1]] = 1
      @max = [@max, e[0], e[1]].max
    end
  end

  def [](x,y)
    if x > y
      @store[x,y]
    elsif x < y
      @store[y,x]
    else
      0
    end
  end

  def []=(x,y,v)
    if x > y
```

```

    @store[x,y]=v
  elsif x < y
    @store[y,x]=v
  else
    0
  end
end

def edge? x,y
  x,y = y,x if x < y
  @store[x,y]==1
end

def add x,y
  @store[x,y] = 1
end

def remove x,y
  x,y = y,x if x < y
  @store[x,y] = 0
  if (degree @max) == 0
    @max -= 1
  end
end

def vmax
  @max
end

def degree x
  sum = 0
  0..@max do |i|
    sum += self[x,i]
  end
  sum
end

def each_vertex
  (0..@max).each {|v| yield v}
end

def each_edge
  for v0 in 0..@max
    for v1 in 0..v0-1
      yield v0,v1 if self[v0,v1]==1
    end
  end
end

end

```

```

mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])

# Wyświetla stopień każdego z wierzchołków (węzłów) grafu: 2 3 2 3
mygraph.each_vertex {|v| puts mygraph.degree(v)}

# Wyświetla listę krawędzi:
mygraph.each_edge do |a,b|
  puts "(#{a},#{b})"
end

# Usuwa pojedynczą krawędź:
mygraph.remove 1,3

# Wyświetla stopień każdego z wierzchołków (węzłów) grafu: 2 2 2 2
mygraph.each_vertex {|v| p mygraph.degree v}

```

Warto pamiętać, że przedstawiona implementacja uniemożliwia reprezentowanie grafów z węzłami połączonymi z samymi sobą, a jedna para węzłów może być ze sobą połączona tylko jedną krawędzią.

Przedstawiona powyżej klasa Graph oferuje możliwość definiowania krawędzi początkowych przez przekazywanie par wartości reprezentujących węzły na wejściu konstruktora. Zaimplementowaliśmy też metody umożliwiające dodawanie i usuwanie krawędzi oraz sprawdzanie istnienia interesujących nas krawędzi. Metoda `vmax` zwraca węzeł grafu charakteryzujący się najwyższym **stopniem** (największą liczbą krawędzi połączonych z danym wierzchołkiem). Metoda `degree` zwraca stopień wskazanego węzła.

I wreszcie klasa Graph implementuje dwa iteratory `each_vertex` i `each_edge`, które odpowiadają za iteracyjne przeszukiwanie odpowiednio krawędzi i wierzchołków grafu.

9.4.2. OKREŚLANIE, CZY WSZYSTKIE WĘZŁY GRAFU SĄ Z NIM POŁĄCZONE

Nie wszystkie grafy składają się wyłącznie z połączonych węzłów. Oznacza to, że przechodząc przez krawędzie, nie zawsze możemy dotrzeć do wszystkich wierzchołków — może się okazać, że niezależnie od wybranej ścieżki graf zawiera wierzchołki nieosiągalne. Łączność z wierzchołkami (tzw. spójność grafu) jest bardzo ważną cechą grafów, stąd konieczność określania, czy badany graf stanowi „jeden kawałek”. Jeśli tak, możemy przyjąć, że każdy węzeł można prędzej czy później osiągnąć niezależnie od węzła początkowego.

Nie będziemy szczegółowo analizowali odpowiedniego algorytmu — Czytelnicy zainteresowani pogłębieniem swojej wiedzy powinni się zaopatrzyć w książkę poświęconą matematyce dyskretnej. Odpowiednią metodę napisaną w języku Ruby przedstawiono na listingu 9.4.

LISTING 9.4. Określanie, czy wszystkie węzły grafu są z nim połączone

```

class Graph

  def connected?
    x = vmax
    k = [x]
    l = [x]
    for i in 0..@max
      l << i if self[x,i]==1
    end
    while !k.empty?
      y = k.shift
      # Możemy teraz odnaleźć wszystkie krawędzie (y,z)
      self.each_edge do |a,b|
        if a==y || b==y
          z = a==y ? b : a
          if !l.include? z
            l << z
            k << z
          end
        end
      end
    end
    end
  end
  if l.size < @max
    false
  else
    true
  end
end

end

mygraph = Graph.new([0,1], [1,2], [2,3], [3,0], [1,3])

puts mygraph.connected?      # true

puts mygraph.euler_path?    # true

mygraph.remove 1,2
mygraph.remove 0,3
mygraph.remove 1,3

puts mygraph.connected?      # false

puts mygraph.euler_path?    # false

```

W przedstawionym kodzie posłużyliśmy się metodą `euler_path?`, o której do tej pory nie wspominaliśmy. Metodę `euler_path?` zdefiniujemy w punkcie 9.4.4 zatytułowanym „Określanie, czy dany graf zawiera cykl Eulera”.

Prezentowany algorytm (po wprowadzeniu kilku nieznaczących modyfikacji) mógłby być z powodzeniem wykorzystywany do wykrywania tzw. **klik** (ang. *cliques*) w grafie, czyli podgrafów, w którym każde dwa wierzchołki są połączone krawędzią.

9.4.3. OKREŚLANIE, CZY DANY GRAF ZAWIERA CYKL EULERA

O żadnej, nawet najbardziej abstrakcyjnej dziedzinie matematyki nie można powiedzieć, że została stworzona w oderwaniu od fenomenu świata rzeczywistego.

— Nikolai Lobachevsky

W pewnych sytuacjach stajemy przed koniecznością stwierdzenia, czy dany graf zawiera tzw. **cykl Eulera** (ang. *Euler circuit*). Wspomniany termin pochodzi od nazwiska matematyka Leonharda Eulera, twórcy dziedziny matematyki nazywanej topologią. (Graf zawierający cykl Eulera bywa czasami nazywany **grafem eulerowskim**, a jego kreślenie nie wymaga odrywania pióra od kartki papieru).

W centrum niemieckiego miasta Königsberg znajdowała się wyspa oddzielona od pozostałych części tego miasta rzeką (przynajmniej na odcinku, w którym rzeka dzieliła się na dwa koryta). Wyspę połączono z resztą miasta siedzioma mostami. Mieszkańcy Königsberga zastanawiali się, czy istnieje możliwość zwiedzenia ich miasta w taki sposób, aby przejść przez każdy most dokładnie raz i znaleźć się na końcu w punkcie wyjścia. W roku 1733 Euler udowodnił, że nie jest to możliwe. Okazało się, że właśnie problem mostów Königsberga stał się jednym z kluczowych elementów oryginalnej teorii grafów.

Problem cyklu Eulera, jak wiele innych zagadek, okazuje się stosunkowo prosty do rozwiązania, dopiero kiedy poznamy odpowiedni mechanizm. Warunkiem koniecznym i wystarczającym do zawierania cyklu Eulera w grafie jest spójność i parzystość **stopnia wszystkich** jego wierzchołków. Poniżej zdefiniowaliśmy prostą metodę weryfikującą tę własność:

```
class Graph

  def euler_circuit?
    return false if !connected?
    for i in 0..@max
      return false if degree(i) % 2 != 0
    end
    true
  end
end

mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])
```

```

flag1 = mygraph.euler_circuit? #false

mygraph.remove 1,3

flag2 = mygraph.euler_circuit? #true

```

9.4.4. OKREŚLANIE, CZY DANY GRAF ZAWIERA ŚCIEŻKĘ EULERA

Ścieżka Eulera (ang. *Euler path*) to nie to samo co cykl Eulera. Cykl Eulera — jak sama nazwa wskazuje — musi się kończyć tam, gdzie się zaczął; konstruując **ścieżkę** Eulera, koncentrujemy się wyłącznie na odwiedzaniu dokładnie raz każdej krawędzi. Poniżej przedstawiono fragment kodu źródłowego, który dobrze ilustruje różnicę dzielącą oba problemy:

```

class Graph

  def euler_path?
    return false if !connected?
    odd=0
    each_vertex do |x|
      if degree(x) % 2 == 1
        odd += 1
      end
    end
    odd <= 2
  end

end

mygraph = Graph.new([0,1],[1,2],[1,3],[2,3],[3,0])

flag1 = mygraph.euler_circuit? #false
flag2 = mygraph.euler_path?   #true

```

9.4.5. NARZĘDZIA UŁATWIAJĄCE OPERACJE NA GRAFACH W JĘZYKU RUBY

Istnieje kilka przydatnych narzędzi wykorzystywanych przez społeczność programistów języka Ruby. Większość tych narzędzi oferuje dość ograniczoną funkcjonalność w obszarze przetwarzania grafów skierowanych i nieskierowanych. Oprogramowania pomocniczego z reguły należy szukać na witrynach internetowych RAA (<http://raa.ruby-lang.org>) oraz Rubyforge (<http://rubyforge.org>). Rozpoznanie interesujących nas narzędzi nie powinno nam sprawić kłopotu, ponieważ większości z nich nadano takie nazwy jak RubyGraph, RGraph czy GraphR.

Czytelnicy zainteresowani doskonałym pakietem GraphViz, który oferuje możliwość wizualizacji nawet najbardziej złożonych grafów zarówno w formie obrazów, jak i drukowalnych dokumentów PostScript, mają do dyspozycji dwa w pełni funkcjonalne interfejsy tego ciekawego produktu. Co więcej, wspomniany pakiet zawiera konstrukcję nazwaną GnomeGraphwidget, która — zgodnie z dokumentacją — „może być wykorzystywana przez aplikacje Ruby Gnome do generowania, wizualizowania i operowania na grafach”. Co prawda nie sprawdzaliśmy rzeczywistej funkcjonalności tego interfejsu, ale warto przy tej okazji wspomnieć, że opisywane rozwiązania ciągle znajdują się w fazie rozwoju sprzed właściwej wersji alfa.

Najkrócej rzecz ujmując, możemy stanąć przed koniecznością skorzystania z któregoś z dostępnych narzędzi. W takim przypadku warto rozważyć samodzielną implementację własnego odpowiednika istniejących narzędzi lub — co w wielu przypadkach jest najlepszym wyjściem — przystąpić do trwających prac nad odpowiednim projektem. Kiedy już opanujemy do perfekcji operacje na grafach, zapewne będziemy się zastanawiali, jak to możliwe, że tak długo radziliśmy sobie bez tych technik.

9.5. KONKLUZJA

W tym rozdziale skoncentrowaliśmy się na istniejącej klasie Set języka Ruby oraz na kilku przykładach samodzielnie budowanych i rozwijanych implementacji struktur danych. Analizując bardziej zaawansowane struktury danych, podjęliśmy próby dziedziczenia po istniejących klasach i stosowania ograniczonej techniki delegacji przez osadzanie egzemplarzy jednych klas w innych klasach. Omówiliśmy metody kreatywnego składowania danych, rozmaite zastosowania dla poszczególnych struktur danych i sposoby tworzenia iteratorów dla naszych niestandardowych klas.

Dokonailiśmy ogólnej analizy stosów i kolejek w kontekście ich zastosowań podczas rozwiązywania rozmaitych problemów. Omówiliśmy też podstawowe cechy takich struktur danych jak drzewa i grafy.

W następnym rozdziale będziemy kontynuować nasze rozważania poświęcone przetwarzaniu danych. Ponieważ jednak dysponujemy już wystarczająco szeroką wiedzą o obiektach składowanych w pamięci operacyjnej, tym razem skoncentrujemy się na innych popularnych technikach składowania danych — w plikach dyskowych (i ogólnie z wykorzystaniem urządzeń wejścia-wyjścia), w bazach danych i w formie obiektów trwałych.