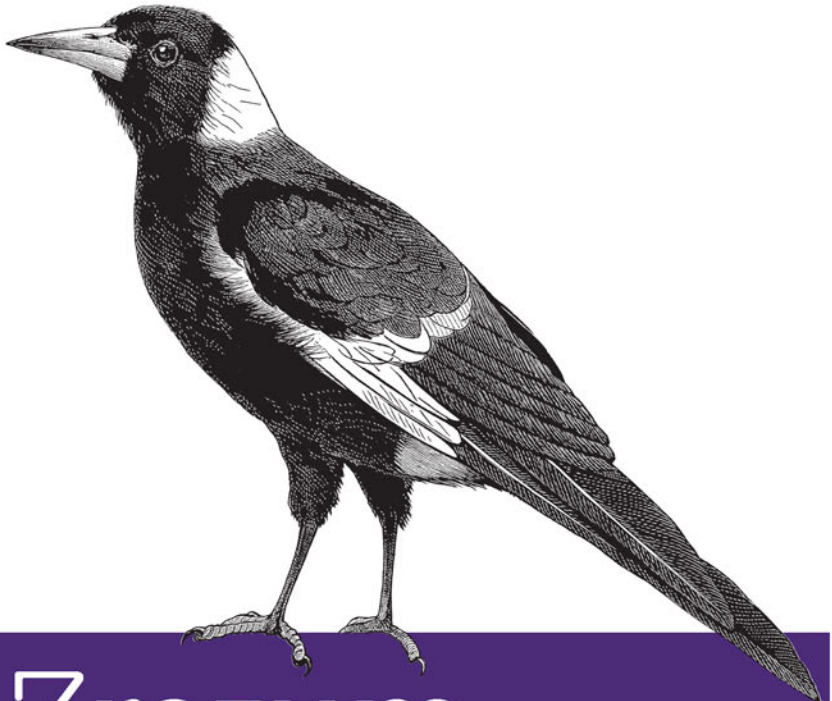
O'REILLY®



Zrozum struktury danych

ALGORYTMY I PRACA NA DANYCH W JAVIE

Helion

Allen B. Downey

Tytuł oryginału: Think Data Structures: Algorithms and Information Retrieval in Java

Tłumaczenie: Łukasz Suma
ISBN: 978-83-283-4092-3

Authorized Polish translation of the English edition of Think Data Structures
ISBN 9781491954386 © 2017 Allen B. Downey

Wydawnictwo HELION
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)

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

Spis treści

Wstęp **7**

1. Interfejsy **13**
Dlaczego są dwa rodzaje list? 14
Interfejsy w języku Java 15
Interfejs List 16
Ćwiczenie 1. 17

2. Analiza algorytmów **21**
Sortowanie przez wybieranie 23
Notacja dużego O 25
Ćwiczenie 2. 26

3. Klasa ArrayList **31**
Klasyfikacja metod klasy MyArrayList 31
Klasyfikacja metody add 33
Wielkość problemu obliczeniowego 36
Powiązane struktury danych 37
Ćwiczenie 3. 39
Uwaga na temat odśmiecania pamięci 42

4. Klasa LinkedList **45**
Klasyfikacja metod klasy MyLinkedList 45
Porównanie klas MyArrayList i MyLinkedList 48
Profilowanie 49
Interpretacja wyników 52
Ćwiczenie 4. 53

5. Lista dwukierunkowa **55**
Wyniki profilowania wydajnościowego 55
Profilowanie metod klasy LinkedList 57
Dodawanie na końcu listy będącej obiektem klasy LinkedList 59

Lista dwukierunkowa 61
Wybór struktury 62

6. Przechodzenie przez drzewo 65
Mechanizmy wyszukiwania 65
Parsowanie kodu HTML 67
Używanie biblioteki jsoup 69
Iterowanie po drzewie DOM 71
Przeszukiwanie w głąb 72
Stosy w języku Java 73
Iteracyjna implementacja DFS 75

7. Dochodzenie do filozofii 77
Pierwsze kroki 77
Interfejsy Iterable i Iterator 78
Klasa WikiFetcher 80
Ćwiczenie 5. 82

8. Indekser 85
Wybór struktury danych 85
Klasa TermCounter 87
Ćwiczenie 6. 90

9. Interfejs Map 95
Implementacja klasy MyLinearMap 95
Ćwiczenie 7. 96
Analiza klasy MyLinearMap 98

10. Mieszanie 101
Mieszanie 101
Jak działa mieszanie? 104
Mieszanie i zmiany 106
Ćwiczenie 8. 107

11. Klasa HashMap 109
Ćwiczenie 9. 109
Analiza klasy MyHashMap 111
Kompromisy 113
Profilowanie klasy MyHashMap 114
Poprawianie klasy MyHashMap 114
Diagramy klas UML 117

12. Klasa TreeMap 119
Co jest nie tak z mieszaniem? 119
Binarne drzewo poszukiwań 120
Ćwiczenie 10. 122
Implementacja klasy TreeMap 124

13. Binarne drzewo poszukiwań 129
Prosta klasa MyTreeMap 129
Wyszukiwanie wartości 130
Implementacja metody put 132
Przechodzenie poprzeczne 133
Metody logarytmiczne 135
Drzewa samorównoważące się 137
Jeszcze jedno ćwiczenie 138

14. Trwałość 139
Redis 140
Serwery i klienty Redisa 141
Tworzenie indeksu przy użyciu Redisa 142
Typy danych Redisa 144
Ćwiczenie 11. 146
Więcej sugestii, z których możesz skorzystać 148
Kilka wskazówek dotyczących projektu 149

15. Pełzanie po Wikipedii 151
Indekser wykorzystujący Redisa 151
Analiza operacji przeglądania 154
Analiza operacji indeksowania 155
Przechodzenie grafu 156
Ćwiczenie 12. 157

16. Wyszukiwanie logiczne 161
Implementacja pełzacza 161
Pozyskiwanie informacji 163
Wyszukiwanie logiczne 164
Ćwiczenie 13. 165
Interfejsy Comparable i Comparator 168
Rozszerzenia 170

17. Sortowanie 173
Sortowanie przez wstawianie 174
Ćwiczenie 14. 176
Analiza sortowania przez scalanie 178
Sortowanie pozycyjne 180
Sortowanie przez kopcowanie 182
Kopiec ograniczony 185
Złożoność pamięciowa 185

Skorowidz 189

ROZDZIAŁ 1.

Interfejsy

W tej książce prezentowane są trzy główne tematy:

Struktury danych
: Korzystając ze struktur oferowanych przez Java Collections Framework (JCF), nauczysz się używać struktur danych takich jak listy i mapy, a także dowiesz się, jak działają.

Analiza algorytmów
: Przedstawiam tu sposoby umożliwiające analizowanie kodu i przewidywanie tego, jak szybko będzie on działał i ile miejsca (pamięci) będzie mu potrzebne.

Pozyskiwanie informacji
: Aby uzasadnić potrzebę zapoznania się z dwoma pierwszymi tematami i uczynić ćwiczenia nieco bardziej interesującymi, skorzystamy z algorytmów i struktur danych w celu zbudowania prostego mechanizmu wyszukiwarki internetowej.

Oto ogólny zarys kolejności poruszonych tematów:

- Zaczniemy od interfejsu `List`, a Twoim zadaniem będzie napisanie klas implementujących go na dwa różne sposoby. Następnie porównamy Twoje implementacje z klasami `ArrayList` i `LinkedList` zapewnianymi przez język Java.
- Następnie przedstawię struktury o charakterze drzewiastym, a Ty opracujesz swoją pierwszą aplikację: program odczytujący strony Wikipedii, analizujący ich treść i poruszający się po utworzonym w ten sposób drzewie w celu wyszukiwania łącz i innych elementów. Skorzystamy z tych narzędzi, aby sprawdzić hipotezę, że większość stron serwisu prowadzi ostatecznie do

strony hasła „filozofia” (informacje na temat tego zjawiska znajdziesz w języku angielskim na stronie *https://en.wikipedia.org/wiki/Wikipedia:↪Getting_to_Philosophy*[1]).

- Kolejna będzie prezentacja interfejsu `Map` i zapewnianej przez język Java implementacji klasy `HashMap`. Następnie napiszesz klasy implementujące ten interfejs za pomocą tablicy mieszającej i binarnego drzewa poszukiwań.
- Na koniec użyjesz tych klas (i kilku innych, które przedstawię po drodze) do zaimplementowania mechanizmu wyszukiwarki internetowej, w tym również pełzacza, który wyszukuje i odczytuje strony, indeksera, który zapisuje zawartość stron internetowych w takiej formie, aby dało się ją efektywnie przeszukiwać, a także pozyskiwacza, który przyjmuje zapytania od użytkownika i zwraca odpowiednie wyniki.

Zacznijmy zatem.

Dlaczego są dwa rodzaje list?

Gdy ludzie zaczynają korzystać z Java Collections Framework, są nieraz zdezorientowani faktem istnienia dwóch różnych klas reprezentujących listę, a mianowicie `ArrayList` i `LinkedList`. Dlaczego Java dostarcza dwie implementacje interfejsu `List`? I na podstawie jakich kryteriów powinno się wybrać tę właściwą do zastosowania w danym przypadku? Odpowiem na te pytania w kilku kolejnych podrozdziałach.

Zacznę od przeglądu `interfejsów` i klas, które je implementują, a także zaprezentuję koncepcję tak zwanego programowania do interfejsu.

W pierwszych kilku ćwiczeniach zaimplementujesz klasy podobne do `ArrayList` i `LinkedList`, aby dowiedzieć się, jak one działają; przekonasz się również, że każda z tych klas ma swoje zalety i wady. Niektóre operacje są wykonywane szybciej lub wymagają mniej pamięci w przypadku zastosowania klasy `ArrayList`, inne są z kolei szybsze lub mniej wymagające pamięciowo, gdy użyje się klasy `LinkedList`. To, która z tych klas jest lepsza w konkretnym zastosowaniu, zależy od tego, jakie operacje są wykonywane częściej.

[1] Niestety w sieci brak podobnego artykułu w języku polskim — *przyp. tłum.*

Interfejsy w języku Java

Interfejs języka Java określa zestaw metod, a każda klasa implementująca dany interfejs musi zapewnić te metody. Oto na przykład kod źródłowy interfejsu Comparable, który został zdefiniowany w pakiecie java.lang:

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

W tej definicji interfejsu używany jest parametr typu T, który powoduje, że interfejs Comparable jest **typem generycznym**, zwanym też **ogólnym** (ang. *generic type*). Aby implementować ten interfejs, klasa musi:

- określać typ, do którego odnosi się T;
- zapewniać metodę o nazwie compareTo, która przyjmuje jako argument obiekt i zwraca wartość int.

Tak na przykład wygląda kod klasy java.lang.Integer:

```
public final class Integer extends Number implements Comparable<Integer> {

   public int compareTo(Integer anotherInteger) {
      int thisVal = this.value;
      int anotherVal = anotherInteger.value;
      return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 : 1));
   }

   // inne metody pominięte
}
```

Klasa ta rozszerza klasę Number, dlatego dziedziczy metody i zmienne instancyjne tej ostatniej i implementuje interfejs Comparable<Integer>, w związku z czym zapewnia metodę compareTo, która przyjmuje jako argument obiekt klasy Integer i zwraca wartość typu int.

Gdy deklaruje się, że klasa implementuje pewien interfejs, kompilator sprawdza, czy zapewnia ona wszystkie metody zdefiniowane w ramach tego interfejsu.

A tak na marginesie: w przedstawionej powyżej implementacji metody compareTo zastosowany został „potrójny operator", zapisywany czasem jako ?:. Jeśli nie jest Ci on znany, odwiedź anglojęzyczną stronę *https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html* [2].

[2] Informacje po polsku na temat trójargumentowego operatora porównania możesz znaleźć na przykład pod adresem *http://notatkiprogramisty.blox.pl/2011/11/Operator-warunkowy-potrojny--ternary-operator.html — przyp. tłum.*

Interfejs List

W ramach Java Collections Framework (JCF) zdefiniowany został interfejs o nazwie `List`; zbiór zapewnia też dwie implementujące go klasy: `ArrayList` i `LinkedList`.

`Interfejs` ten definiuje to, co oznacza bycie listą; każda implementująca go klasa musi zapewnić określony zestaw metod, w tym metody `add`, `get`, `remove`, a także około 20 innych.

Klasy `ArrayList` i `LinkedList` oferują te metody, dlatego mogą być używane wymiennie. Metoda napisana w taki sposób, aby operować na obiekcie typu `List`, poradzi sobie z obiektem klasy `ArrayList`, `LinkedList`, a także z każdym innym, którego klasa implementuje interfejs `List`.

Oto wymyślony przykład, który demonstruje sposób działania tego mechanizmu:

```
public class ListClientExample {
   private List list;

   public ListClientExample() {
      list = new LinkedList();
   }

   private List getList() {
      return list;
   }

   public static void main(String[] args) {
      ListClientExample lce = new ListClientExample();
      List list = lce.getList();
      System.out.println(list);
   }
}
```

Klasa `ListClientExample` nie robi nic użytecznego, ale ma wszystkie podstawowe elementy klasy, która **hermetyzuje** interfejs `List`. Oznacza to, że zawiera ona zmienną instancyjną typu `List`. Skorzystam z tej klasy, aby przedstawić ideę, a następnie zajmiesz się pierwszym ćwiczeniem.

Konstruktor klasy `ListClientExample` inicjalizuje składnik `list` poprzez **utworzenie instancji** (czyli **egzemplarza**) klasy `LinkedList`. Metoda gettera o nazwie `getList` zwraca referencję do wewnętrznego obiektu typu `List`, a metoda `main` zawiera kilka wierszy kodu, których zadaniem jest przetestowanie pozostałych metod.

Ważne w tym przykładzie jest to, że wykorzystujemy w nim typ `List` wszędzie tam, gdzie jest to możliwe, i unikamy precyzowania, czy chodzi nam o klasę `LinkedList` czy o klasę `ArrayList`, dopóki nie jest to konieczne. Zmienna instancyjna jest na przykład zadeklarowana jako `List`, a metoda `getList` zwraca obiekt typu `List`, w żadnym z tych przypadków nie określamy jednak dokładnie, o jaki dokładnie rodzaj listy nam chodzi.

Jeśli w pewnym momencie zmienisz zdanie i postanowisz zastosować klasę `ArrayList`, będziesz jedynie musiał zmienić konstruktor; żadne inne zmiany nie będą konieczne.

Ten styl określany jest mianem **programowania opartego na interfejsach** (ang. *interface-based programming*) lub — nieco bardziej potocznie — programowania do interfejsu (ang. *programming to an interface*). Więcej informacji po angielsku na temat tej techniki znajdziesz pod adresem *https://en.wikipedia.org/wiki/Interface-based_programming*[3]. Mówimy tutaj o ogólnej idei interfejsu, nie konkretnie o `interfejsie` języka Java.

Gdy korzystasz z biblioteki, Twój kod powinien być uzależniony wyłącznie od jej interfejsu, takiego jak `List`. Kod nie powinien zależeć od konkretnej implementacji, takiej jak `ArrayList`. Dzięki temu, jeśli implementacja ulegnie w przyszłości zmianie, wykorzystujący ją kod nadal będzie działał bez problemu.

Z drugiej jednak strony, jeśli zmieni się interfejs, zmodyfikowany będzie musiał zostać również zależny od niego kod. To właśnie z tego powodu programiści tworzący biblioteki unikają zmian interfejsów, gdy nie są absolutnie konieczne.

Ćwiczenie 1.

Jako że jest to pierwsze ćwiczenie w tej książce, będzie ono bardzo proste. Chodzi w nim o to, aby wziąć kod przedstawiony w poprzednim podrozdziale i **zamienić implementację**. Należy zatem zastąpić odwołanie do klasy `LinkedList` odwołaniem do klasy `ArrayList`. Dzięki temu, że nasz kod powstał zgodnie z założeniem programowania do interfejsu, przełączenie implementacji będzie wymagało zmiany tylko jednego wiersza i dodania instrukcji `import`.

[3] Informacje na ten temat w języku polskim znaleźć można na przykład na stronie *https://javastart.pl/static/programowanie-obiektowe/polimorfizm/ — przyp. tłum.*

Zacznij od konfiguracji swojego środowiska programistycznego. W przypadku wszystkich ćwiczeń przedstawionych w tej książce będzie Ci potrzebna możliwość kompilacji i uruchamiania kodu napisanego w języku Java. Opracowałem przykłady, korzystając ze środowiska Java SE Development Kit 7. Jeśli korzystasz z nowszej wersji rozwiązania, wszystko w dalszym ciągu powinno działać. Jeśli używasz starszej wersji środowiska, możesz natknąć się na pewne niezgodności.

Zalecam Ci zastosowanie interaktywnego środowiska programistycznego (ang. *interactive development environment IDE*), które zapewnia możliwości sprawdzania składni, autouzupełniania oraz refaktoringu kodu źródłowego. Funkcje te pomogą Ci unikać błędów i szybko znajdować te, których uniknąć Ci się nie uda. Jeśli jednak przygotowujesz się na techniczną rozmowę kwalifikacyjną, musisz pamiętać, że w jej trakcie nie będziesz mieć do dyspozycji wszystkich tych narzędzi, dlatego może Ci się przydać trochę praktyki w pisaniu kodu bez ich użycia.

Jeśli nie pobrałeś jeszcze kodu związanego z tą książką, zapoznaj się ze wskazówkami przedstawionymi w podrozdziale wstępu zatytułowanym „Praca z kodem".

W katalogu o nazwie `code` powinieneś znaleźć następujące elementy:

- `build.xml` to plik Anta, który ułatwia kompilację i uruchamianie kodu;
- `lib` to katalog zawierający biblioteki, których będziesz potrzebować (w przypadku tego ćwiczenia będzie to jedynie JUnit);
- `src` to katalog zawierający kod źródłowy.

Gdy przejdziesz do katalogu `src/com/allendowney/thinkdast`, znajdziesz kod źródłowy dla tego ćwiczenia:

- Plik `ListClientExample.java` zawiera kod przedstawiony w poprzednim podrozdziale.
- Plik `ListClientExampleTest.java` zawiera opracowany za pomocą biblioteki JUnit test klasy `ListClientExample`.

Przejrzyj kod klasy `ListClientExample` i upewnij się, że rozumiesz jego działanie. Następnie go skompiluj i uruchom. Jeśli używasz Anta, przejdź do katalogu `code` i wywołaj polecenie **`ant ListClientExample`**.

Niewykluczone, że w wyniku tego na ekranie pojawi się ostrzeżenie podobne do następującego:

```
List is a raw type. References to generic type List<E>
should be parameterized.[4]
```

Aby uprościć przykład, nie zawracałem sobie głowy określaniem typu elementów przechowywanych za pomocą obiektu klasy `List`. Jeśli ostrzeżenie to Ci przeszkadza, możesz się go pozbyć, zastępując każde wystąpienie typu `List` lub `LinkedList` odpowiednio konstrukcją `List<Integer>` lub `LinkedList<Integer>`.

Przejrzyj kod klasy `ListClientExampleTest`. Jest w niej wykonywany pojedynczy test, w ramach którego tworzy się obiekt klasy `ListClientExample`, wywołuje się metodę `getList`, a następnie sprawdza się, czy uzyskany wynik jest typu `ArrayList`. Początkowo test ten zakończy się niepowodzeniem, ponieważ wynik jest obiektem klasy `LinkedList`, a nie `ArrayList`. Uruchom ten test i przekonaj się, że się nie powiedzie.

Uwaga: Test ten ma sens w przypadku tego ćwiczenia, nie jest jednak dobrym przykładem testu jako takiego. Dobre testy powinny sprawdzać, czy testowana klasa spełnia wymagania stawiane przez *interfejs*; nie powinny być uzależnione od szczegółów *implementacji*.

W klasie `ListClientExample` zastąp typ `LinkedList` typem `ArrayList`. Być może będzie też trzeba dodać instrukcję `import`. Skompiluj i uruchom kod klasy `ListClientExample`. Potem ponownie uruchom odpowiedni test. Dzięki dokonanej przez Ciebie zmianie test tym razem powinien się powieść.

Aby do tego doprowadzić, musiałeś jedynie zmienić jeden wiersz konstruktora klasy; nie było potrzeby wprowadzania jakichkolwiek innych zmian w którymkolwiek miejscu, w jakim występuje typ `List`. A co stanie się, gdy to zrobisz? Śmiało, spróbuj: zastąp jedno lub wszystkie wystąpienia słowa `List` słowem `ArrayList`. Kiedy to zrobisz, program w dalszym ciągu powinien działać poprawnie, będzie jednak „nadmiernie skonkretyzowany”. Jeśli zmienisz w przyszłości zdanie i postanowisz ponownie przełączyć implementację, będziesz musiał zmienić większą ilość kodu.

Co się stanie, gdy w konstruktorze klasy `ListClientExample` zmienisz typ `Array`
`↪List` na typ `List`? Dlaczego nie możesz utworzyć egzemplarza tego typu?

[4] `List` to typ surowy. Odwołania do generycznego typu `List<E>` powinny być sparametryzowane — *przyp. tłum.*

Skorowidz

A

algorytm
  DFS, 75
  kwadratowy, 22
  liniowy, 22
  logarytmiczny, 122
  stałoczasowy, 22
analiza
  algorytmów, 21
  operacji indeksowania, 155
  operacji przeglądania, 154
  z amortyzacją, 35
API, application programming interface, 9

B

biblioteka jsoup, 69
binarne drzewo poszukiwań, 120, 129

D

diagramy klas UML, 117
drzewo, 120, 129
  DOM, 67, 71
  samorównoważące się, 137

E

egzemplarz klasy, 16

F

FIFO, 73
funkcja
  mieszająca, 102
  skrótu, 102

G

graf, 156

H

haszowanie, *Patrz* mieszanie, 101, 104
HTML, 67

I

identyfikator, 135
iloczyn zbiorów, 86
indeks, 85
indekser, 151
indeksowanie, 65, 155
instancja, 16
interfejs, 15
  Comparable, 168
  Comparator, 168
  Iterable, 78
  Iterator, 78
  List, 16
  Map, 95
  programowania aplikacji, API, 9

K

klasa
ArrayList, 31
HashMap, 109
LinkedList, 45
ListNode, 38
MyArrayList, 48
MyHashMap, 111, 114
MyLinearMap, 95, 98
MyLinkedList, 48
MyTreeMap, 129
TermCounter, 87
TreeMap, 119, 124
WikiFetcher, 80
klasy anonimowe, 50
klucz, 86
kopiec ograniczony, 185

L

LIFO, 73
lista, 14
dodawanie elementów, 59
dwukierunkowa, 55, 61

M

mapa, 86
metoda
add, 33
get, 31
getElementById, 70
indexOf, 32
keySet, 133
put, 132
remove, 47
select, 70
set, 32
metody
klasy MyArrayList, 31
klasy MyLinkedList, 45
logarytmiczne, 135
opakowujące, 88
mieszanie, 101, 104

N

notacja dużego O, 25

O

odśmiecanie pamięci, 42

P

parametr typu T, 15
parsowanie kodu HTML, 67
pełzacz, 161
pełzanie, 65
pozyskiwanie informacji, 66, 163
profilowanie, 21, 49
klasy LinkedList, 57
klasy MyHashMap, 114
wydajnościowe, 55
programowanie oparte na interfejsach, 17
przechodzenie poprzeczne, 133
przeglądanie, 154
przeszukiwanie w głąb, 71

R

Redis, 140
indekser, 151
klienty, 141
serwery, 141
tworzenie indeksu, 142
typy danych, 144
rząd wzrostu, 26

S

schemat obiektów listy, 39
skrót, 102
sortowanie, 173
  pozycyjne, 180
  przez kopcowanie, 182
  przez scalanie, 178
  przez wstawianie, 174
  przez wybieranie, 23
stos, 73
struktura danych, 37, 85

T

tworzenie
  indeksu, 142
  instancji, 16
typ generyczny, 15
typy danych Redisa, 144

U

UML, 117

W

wyrażenia regularne, 89
wyszukiwanie
  logiczne, 164
  wartości, 130
wyszukiwarka internetowa, 65

Z

złożoność pamięciowa, 185
znacznik, 67

O'REILLY®

Myśl jak informatyk i zrozum algorytmy!

By stać się dobrym programistą, musisz przyswoić najważniejsze elementy inżynierii oprogramowania: algorytmy i struktury danych. Nie są to zagadnienia proste i z pewnością niejednego studenta informatyki kosztowały wiele zarwanych nocy. Niestety, istniejące na rynku książki dotyczące tych zagadnień nie ułatwiają nauki. Najczęściej są przeładowane matematycznymi wywodami, zbyt teoretyczne i oderwane od konkretnych zastosowań.

Jeśli postanowiłeś zdobyć praktyczną wiedzę o algorytmach i strukturach danych, a przy tym nieźle posługujesz się Javą, to trzymasz w rękach właściwą publikację. Znajdziesz tu jedynie niezbędną teorię, a przede wszystkim będziesz zajmować się analizą implementacji algorytmów, mierzeniem ich wydajności oraz kontrolą wersji i testami jednostkowymi. Mimo niewielkiej objętości w książce znalazły się również ambitniejsze zagadnienia, na przykład trwałe struktury danych tworzone przez bazy danych Redis. W każdym rozdziale zamieszczono praktyczne ćwiczenia wraz z odpowiednim kodem testującym.

W tej książce między innymi:

- wprowadzenie do interfejsów Javy
- analiza algorytmów
- binarne drzewo przeszukiwania
- wyszukiwanie logiczne
- sortowanie

Allen B. Downey jest profesorem informatyki na uczelni Olin College of Engineering. Wykładał na Wellesley College, Colby College i Uniwersytecie Kalifornijskim w Berkeley. W 2009 roku pracował jako badacz dla firmy Google. Doktorat z dziedziny informatyki obronił na Uniwersytecie Kalifornijskim w Berkeley. Jest autorem wielu książek i artykułów publikowanych w renomowanych periodykach. Zawodowo interesuje się statystyką Bayesa i nauką o prawdopodobieństwie.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-4092-3
9 788328 340923

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 39,90 zł