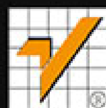
TWÓJ PRZEWODNIK PO JĘZYKU JAVA

Apress®

Java
Zaawansowane zastosowania

Noel Kalicharan



Helion

Tytuł oryginału: Advanced Topics in Java: Core Concepts in Data Structures

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-9426-6

Original edition copyright © 2014 by Noel Kalicharan.
All rights reserved.

Polish edition copyright © 2014 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

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

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

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

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)

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

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorach 9

Recenzenci techniczni 11

Wprowadzenie 13

Rozdział 1. Sortowanie, przeszukiwanie i scalanie 15

1.1. Sortowanie tablic: sortowanie przez wybieranie 15
1.2. Sortowanie tablic: sortowanie przez wstawianie 19
1.3. Wstawianie elementu w odpowiednim miejscu 26
1.4. Sortowanie tablicy łańcuchów znaków 27
1.5. Sortowanie tablic równoległych 29
1.6. Wyszukiwanie binarne 30
1.7. Przeszukiwanie tablicy łańcuchów znaków 32
1.8. Przykład: zliczanie wystąpień wyrazów 34
1.9. Scalanie posortowanych list 37
Ćwiczenia 40

Rozdział 2. Wprowadzenie do obiektów 43

2.1. Obiekty 44
2.2. Definiowanie klas i tworzenie obiektów 44
2.3. Konstruktory 48
2.4. Hermetyzacja danych, metody akcesorów i mutatorów 51
2.5. Wyświetlanie danych obiektów 55
2.6. Klasa Part 57
2.7. Jakie nazwy nadawać plikom źródłowym? 59
2.8. Stosowanie obiektów 60
2.9. Wskaźnik null 63
2.10. Przekazywanie obiektu jako argumentu 64
2.11. Tablice obiektów 65
2.12. Przeszukiwanie tablicy obiektów 67
2.13. Sortowanie tablicy obiektów 70
2.14. Zastosowanie klasy do grupowania danych: licznik występowania słów 71
2.15. Zwracanie więcej niż jednej wartości: głosowanie 74
Ćwiczenia 80

Kup ksi k Pole ksi k

Rozdział 3. Listy powiązane ... 83
3.1. Definiowanie list powiązanych ... 83
3.2. Proste operacje na listach powiązanych ... 85
3.3. Tworzenie listy powiązanej: dodawanie elementów na końcu listy ... 88
3.4. Wstawianie elementów do list powiązanych ... 91
3.5. Tworzenie listy powiązanej: dodawanie elementu na początku listy ... 93
3.6. Usuwanie elementów z list powiązanych ... 94
3.7. Tworzenie posortowanej listy powiązanej ... 95
3.8. Klasa listy powiązanej ... 99
3.9. Jak zorganizować pliki źródłowe Javy? ... 104
3.10. Rozszerzanie klasy LinkedList ... 106
3.11. Przykład: palindromy ... 107
3.12. Zapisywanie listy powiązanej ... 111
3.13. Tablice a listy powiązane ... 111
3.14. Przechowywanie list powiązanych przy użyciu tablic ... 112
3.15. Scalanie dwóch posortowanych list powiązanych ... 113
3.16. Listy cykliczne i dwukierunkowe ... 116
Ćwiczenia ... 120

Rozdział 4. Stosy i kolejki ... 123
4.1. Abstrakcyjne typy danych ... 123
4.2. Stosy ... 124
4.3. Ogólny typ Stack ... 130
4.4. Konwertowanie wyrażenia z zapisu wrostkowego na przyrostkowy ... 134
4.5. Kolejki ... 142
Ćwiczenia ... 151

Rozdział 5. Rekurencja ... 153
5.1. Definicje rekurencyjne ... 153
5.2. Pisanie funkcji rekurencyjnych w języku Java ... 154
5.3. Konwersja liczby dziesiątkowej na dwójkową przy użyciu rekurencji ... 156
5.4. Wyświetlanie listy powiązanej w odwrotnej kolejności ... 159
5.5. Problem wież Hanoi ... 160
5.6. Funkcja podnosząca liczbę do potęgi ... 162
5.7. Sortowanie przez scalanie ... 163
5.8. Zliczanie organizmów ... 166
5.9. Odnajdywanie drogi przez labirynt ... 170
Ćwiczenia ... 174

Rozdział 6. Liczby losowe, gry i symulacje ... 177
6.1. Liczby losowe ... 177
6.2. Liczby losowe i pseudolosowe ... 178
6.3. Komputerowe generowanie liczb losowych ... 179
6.4. Zgadywanka ... 180
6.5. Ćwiczenia z dodawania ... 181
6.6. Gra Nim ... 182
6.7. Rozkłady nierównomierne ... 186
6.8. Symulowanie realnych problemów ... 189
6.9. Symulacja kolejki ... 190
6.10. Szacowanie wartości liczbowych przy użyciu liczb losowych ... 193
Ćwiczenia ... 196

Kup ksi k Pole ksi k

Rozdział 7. Praca z plikami ... 199
7.1. Operacje wejścia-wyjścia w Javie ... 199
7.2. Pliki tekstowe i binarne ... 200
7.3. Wewnętrzne i zewnętrzne nazwy plików ... 200
7.4. Przykład: porównywanie dwóch plików ... 201
7.5. Konstrukcja try…catch ... 202
7.6. Operacje wejścia-wyjścia na plikach binarnych ... 205
7.7. Pliki o dostępie swobodnym ... 209
7.8. Pliki indeksowane ... 213
7.9. Aktualizacja pliku o dostępie swobodnym ... 221
Ćwiczenia ... 224

Rozdział 8. Wprowadzenie do zagadnień drzew binarnych ... 225
8.1. Drzewa ... 225
8.2. Drzewa binarne ... 227
8.3. Przechodzenie drzew binarnych ... 228
8.4. Sposoby reprezentacji drzew binarnych ... 231
8.5. Budowanie drzewa binarnego ... 233
8.6. Binarne drzewa poszukiwań ... 237
8.7. Budowanie binarnego drzewa poszukiwań ... 240
8.8. Budowanie drzew binarnych ze wskaźnikami rodzica ... 244
8.9. Przechodzenie drzewa poziomami ... 249
8.10. Użyteczne funkcje operujące na drzewach binarnych ... 254
8.11. Usuwanie wierzchołków z binarnego drzewa poszukiwań ... 255
8.12. Tablice jako sposób reprezentacji drzew binarnych ... 257
Ćwiczenia ... 260

Rozdział 9. Zaawansowane metody sortowania ... 263
9.1. Sortowanie przez kopcowanie ... 263
9.2. Budowanie kopca przy użyciu metody siftUp ... 269
9.3. Analiza algorytmu sortowania przez kopcowanie ... 272
9.4. Kopce i kolejki priorytetowe ... 273
9.5. Sortowanie listy elementów przy użyciu sortowania szybkiego ... 274
9.6. Sortowanie Shella (z użyciem malejących odstępów) ... 284
Ćwiczenia ... 288

Rozdział 10. Haszowanie ... 291
10.1. Podstawy haszowania ... 291
10.2. Rozwiązanie problemu wyszukiwania i wstawiania przy użyciu haszowania ... 292
10.3. Rozwiązywanie kolizji ... 297
10.4. Przykład: licznik występowania słów ... 307
Ćwiczenia ... 310

Skorowidz ... 313

Kup ksi k

Pole ksi k

Kup ksi k

Pole ksi k

ROZDZIAŁ 9

Zaawansowane metody sortowania

W tym rozdziale wyjaśnimy takie zagadnienia jak:

- sterta oraz algorytm sortowania przez kopcowanie przy użyciu metody `siftDown`,
- tworzenia kopca za pomocą metody `siftUp`,
- analiza wydajności algorytmu sortowania przez kopcowanie,
- użycie kopca w celu implementacji kolejki priorytetowej,
- sortowanie listy elementów z wykorzystaniem algorytmu sortowania szybkiego,
- odnajdywanie *k*-tego najmniejszego elementu listy,
- sortowanie listy elementów z zastosowaniem algorytmu sortowania Shella (z użyciem malejących odstępów).

W rozdziale 1. przedstawiono dwie proste metody sortowania (sortowanie przez wybieranie oraz przez wstawianie), pozwalające na sortowanie listy elementów. W tym rozdziale dokładnie przeanalizujemy kilka innych, bardziej wydajnych metod sortowania — sortowanie przez kopcowanie (ang. *heapsort*), sortowanie szybkie (ang. *quicksort*) oraz sortowanie Shella (ang. *shellsort*).

9.1. Sortowanie przez kopcowanie

Sortowanie przez kopcowanie (ang. *heapsort*) to algorytm sortowania, który *interpretuje* elementy w tablicy jako prawie kompletne drzewo binarne. Przeanalizujmy następującą tablicę, którą należy posortować w kolejności rosnącej.

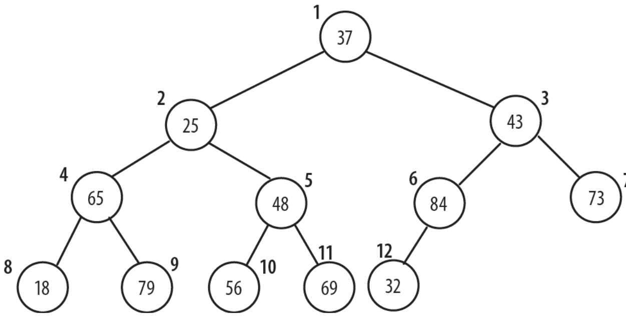
num

37	25	43	65	48	84	73	18	79	56	69	32
1	2	3	4	5	6	7	8	9	10	11	12

Taką tablicę możemy potraktować jako prawie kompletne drzewo binarne składające się z 12 wierzchołków, które zostało przedstawione na rysunku 9.1.

Załóżmy, że teraz postawimy wymóg, by wartości w każdym z wierzchołków były większe lub równe wartościom w lewym i prawym poddrzewie; zakładamy przy tym, że nie są one puste. Krótko mówiąc, zobaczymy, w jaki sposób można przeorganizować wierzchołki tak, by *każdy z nich* spełniał tę właściwość. Zanim to zrobimy, nadamy takiej strukturze danych nazwę kopca.

Kup ksi k Pole ksi k



Rysunek 9.1. *Zawartość tablicy przedstawiona w formie drzewa binarnego*

> ***Kopcem*** *nazywamy prawie kompletne drzewo binarne, takie że wartość korzenia tego drzewa jest większa lub równa wartościom jego lewego i prawego dziecka, a prawe i lewe poddrzewo także są kopcami.*

Natychmiastową konsekwencją takiej definicji jest to, że największa wartość w drzewie znajduje się w jego korzeniu. Taki kopiec jest nazywamy **kopcem maksymalnym** (ang. *max-heap*). W podobny sposób możemy zdefiniować **kopiec minimalny** — wystarczy zamienić relację większości na relację mniejszości. A zatem w kopcu minimalnym jego korzeń zawiera wartość *najmniejszą*.

Spróbujmy teraz zmodyfikować drzewo binarne z rysunku 9.1, by stało się kopcem maksymalnym.

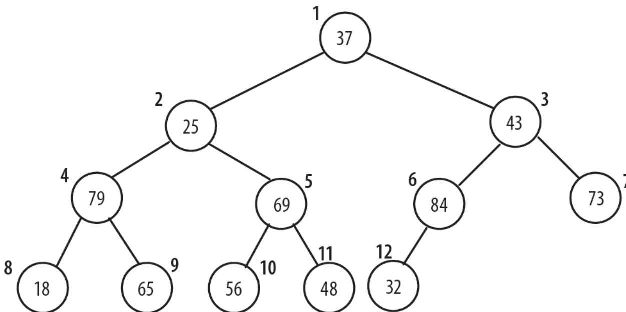
9.1.1. Konwersja drzewa binarnego w kopiec maksymalny

Najpierw należy zauważyć, że wszystkie liście są kopcami, gdyż nie mają żadnych dzieci.

Zaczynając od ostatniego wierzchołka, który nie jest liściem (w naszym przykładzie jest to wierzchołek numer 6), przekształcimy drzewo, którego jest on korzeniem, w kopiec maksymalny. Jeśli wartość wierzchołka jest większa od wartości jego dzieci, nie musimy nic robić. Tak właśnie jest w przypadku wierzchołka 6., gdyż 84 jest większe od 32.

Następnie przechodzimy do wierzchołka o numerze 5. W jego przypadku wartość 48 jest mniejsza od wartości przynajmniej jednego dziecka (w właściwie, od obu dzieci, gdyż ich wartości wynoszą odpowiednio 56 i 69). Najpierw znajdujemy większe dziecko (69) i zamieniamy jego zawartość z zawartością wierzchołka 5. W efekcie liczba 69 zostaje zapisana w wierzchołku 5., a liczba 48 w wierzchołku 11.

Następnie przechodzimy do wierzchołka 4. Większe z jego dzieci, 79, zostaje przeniesione do wierzchołka 4., a 65 do wierzchołka 9. Po zakończeniu tego etapu prac drzewo ma postać przedstawioną na rysunku 9.2.



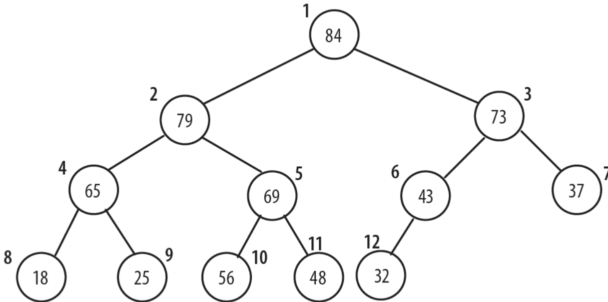
Rysunek 9.2. *Drzewo po przetworzeniu wierzchołków 6., 5. oraz wierzchołka numer 4*

Kup ksi k   Pole ksi k

Teraz przechodzimy do wierzchołka 3. W jego przypadku konieczne jest przeniesienie liczby 43. Większą z wartości dzieci tego wierzchołka jest 84, zatem zamieniamy wartości w wierzchołkach 3. i 6. Aktualna wartość wierzchołka 6. (43) jest większa od wartości jego dziecka (32), więc nie musimy robić nic więcej. Trzeba jednak zwrócić uwagę, że gdyby wartością wierzchołka 6. była liczba (przykładowo) 28, musielibyśmy zamienić ją miejscami z liczbą 32.

Po przejściu do wierzchołka 2. okazuje się, że konieczne jest zamienienie umieszczonej w nim liczby 28 z wartością większego z dzieci, czyli liczbą 79. Kiedy to zrobimy, liczba 25 umieszczona w wierzchołku 4. będzie mniejsza od liczby 65 w jego prawym dziecku, czy wierzchołku numer 9. Zatem także te dwie liczby należy zamienić miejscami.

I w końcu, po przejściu do wierzchołka 1. zamieniamy jego zawartość, 37, z zawartością większego z jego dzieci, czyli z liczbą 84. Następnie jest ona dalej zamieniana z (nowym) większym dzieckiem zawierającym liczbę 73. W ten sposób drzewo staje się kopcem, a jego ostateczną postać przedstawiono na rysunku 9.3.



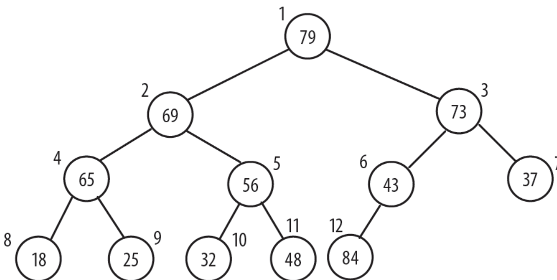
Rysunek 9.3. *Ostateczna postać drzewa, które już jest kopcem*

9.1.2. Proces sortowania

Warto zwrócić uwagę, że po przekształceniu w kopiec korzeń drzewa zawiera największą z jego wartości, 84. Skoro wartości zapisane w tablicy tworzą kopiec, zatem teraz możemy je posortować w kolejności rosnącej. Oto sposób, w jaki należy to zrobić.

- Musimy zapisać ostatni element, czyli 32, w jakimś tymczasowym miejscu i przenieść liczbę 84 na ostatnie miejsce (do wierzchołka numer 12), zwalniając tym samym wierzchołek numer 1. Teraz musimy wyobrazić sobie, że liczba 32 znajduje się w wierzchołku 1. i przenieść ją tak, by elementy tablicy z zakresu od 1 do 11 tworzyły kopiec. Można to zrobić w następujący sposób.
- 32 jest zamieniane z większym z jego dzieci, czyli liczbą 79, która zostaje przeniesiona do wierzchołka 1. Liczba 32 jest dalej zamieniana z jej (nowym) większym dzieckiem, którym jest liczba 69; co sprawia, że liczba 32 trafia do wierzchołka 2.

W końcu liczba 32 zostaje zamieniona z liczbą 56, co sprawia, że drzewo przyjmuje postać przedstawioną na rysunku 9.4.



Rysunek 9.4. *Postać drzewa po umieszczeniu wartości 84 w odpowiednim miejscu i zreorganizowaniu drzewa*

Kup ksi k Pole ksi k

Na tym etapie sortowania druga największa liczba, 79, znajduje się w wierzchołku 1. Umieszczamy ją zatem w wierzchołku 11., a liczba 48 zostaje „przesiana w dół" z wierzchołka 1., tak by wierzchołki od 1. do 10. tworzyły kopiec. Teraz trzecią co do wielkości liczbą umieszczoną w drzewie jest 73, liczba ta znajduje się w jego korzeniu. W kolejnym etapie sortowania umieszczamy ją w wierzchołku 10. itd. Taki proces jest kontynuowany, aż do momentu posortowania całej tablicy.

Po początkowym utworzeniu kopca proces jego sortowania można opisać, posługując się pseudokodem, w następujący sposób.

```
for k = n downto 2 do
   item = num[k]        // pobieramy aktualnie ostatni element
   num[k] = num[1]      // przenosimy wierzchołek kopca do jego ostatniego wierzchołka
   siftDown(item, num, 1, k-1)   // odtwarzamy właściwości kopca w zakresie od 1 do k–1
endfor
```

Przy czym metoda `siftDown(item, num, 1, k-1)` przyjmuje, że spełnione są następujące założenia:

- element `num[1]` jest pusty,
- elementy `num[2]` do `num[k-1]` tworzą kopiec.

Zaczynamy od pozycji numer 1: wartość `item` jest wstawiana w taki sposób, że `num[1]` do `num[k-1]` tworzą kopiec.

W opisanym powyżej procesie sortowania podczas każdej iteracji pętli wartość z aktualnie ostatniej pozycji (`k`) jest zapisywana w zmiennej `item`. Wartość z wierzchołka 1. jest przenoszona na pozycję `k`, wierzchołek 1. zostanie opróżniony (i dostępny), a wszystkie wierzchołki z zakresu od `2` do `k-1` spełniają warunek kopca.

Wywołanie `siftDown(item, num, 1, k-1)` doda wartość w zmiennej `item` do tablicy w taki sposób, że elementy `num[1]` do `num[k-1]` będą tworzyć kopiec. Dzięki temu zapewnimy, że kolejna największa wartość w kopcu znajdzie się w wierzchołku 1.

Bardzo użyteczną cechą metody `siftDown` (kiedy już ją napiszemy) będzie to, że przy jej użyciu będziemy w stanie utworzyć początkowy kopiec z przekazanej tablicy. Przypomnijmy sobie proces tworzenia kopca opisany w punkcie 9.1.1. Dla każdego wierzchołka (dajmy na to `h`) „przesiewamy wartość w dół", tak by utworzyć kopiec o korzeniu w wierzchołku `h`. Aby zastosować metodę `siftDown` w obecnej sytuacji, musimy uogólnić ją w następujący sposób:

```
void siftDown(int key, int num[], int root, int last)
```

Metoda ta zakłada, że spełnione są następujące warunki:

- element `num[root]` jest pusty,
- indeks `last` wskazuje ostatni element tablicy `num`,
- element `num[root*2]`, jeśli istnieje (`root*2` ≤ `last`), jest korzeniem kopca,
- element `num[root*2+1]`, jeśli istnieje (`root*2+1` ≤ `last`), jest korzeniem kopca.

Zaczynamy od elementu o indeksie `root`: wartość `key` jest wstawiana do tablicy `num` w taki sposób, że `num[root]` będzie korzeniem kopca.

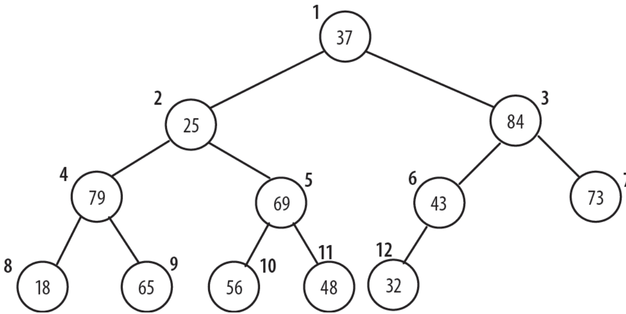
Dysponując tablicą wartości `num[1]` do `num[n]`, możemy utworzyć kopiec, postępując w sposób opisany następującym pseudokodem.

```
for h = n/2 downto 1 do        // n/2 jest ostatnim wierzchołkiem, który nie jest liściem
   siftDown(num[h], num, h, n);
```

Teraz zajmiemy się napisaniem metody `siftDown`.

Przeanalizujmy kopiec przedstawiony na rysunku 9.5.

Kup ksi k   Pole ksi k



Rysunek 9.5. *Kopiec, z wyjątkiem wierzchołków o numerach 1 i 2*

Wszystkie wierzchołki, z wyjątkiem 1. i 2., spełniają warunki kopca, czyli są większe od swoich dzieci lub im równe. Załóżmy, że chcemy sprawić, by wierzchołek 2. stał się korzeniem kopca. Aktualnie umieszczona w nim liczba 25 jest mniejsza od jego dzieci (którymi są liczby 79 i 69). Chcemy zatem napisać metodę `siftDown` tak, by poniższe wywołanie doprowadziło do utworzenia kopca:

```
siftDown(26, num, 2, 12)
```

W powyższym wywołaniu `25` jest wartością parametru `key`, `num` jest tablicą, `2` jest indeksem korzenia, a `12` indeksem ostatniego przetwarzanego elementu tablicy.

Po zakończeniu wywołania każdy z wierzchołków, od 2. do 12., będzie korzeniem kopca, a następujące wywołanie sprawi, że cała tablica będzie kopcem:

```
siftDown(37, num, 1, 12)
```

Zarys działania metody `siftDown` można opisać w następujący sposób.

```
znajdujemy większe z dzieci wierzchołka num[root]; //załóżmy, że jest to wierzchołek m
if (key >= num[m]) gotowe; zapisujemy key w num[root]
//wartość key jest mniejsza od większego z dzieci
zapisujemy num[m] w num[root]   // wybieramy większe z dzieci
ustawiamy root na m
```

Ten proces jest powtarzamy tak długo, jak długo wartość wierzchołka `root` jest większa od wartości jego dzieci bądź też wierzchołek ten nie będzie mieć dzieci. Poniżej przedstawiono kod metody `siftDown`.

```
public static void siftDown(int key, int[] num, int root, int last) {
   int bigger = 2 * root;
   while (bigger <= last) { //dopóki jest co najmniej jedno dziecko
      if (bigger < last) //istnieje także prawe dziecko; znajdujemy większe
         if (num[bigger+1] > num[bigger]) bigger++;
      //'bigger' zawiera indeks większego dziecka
      if (key >= num[bigger]) break;
      //wartość key jest mniejsza; wybieramy num[bigger]
      num[root] = num[bigger];
      root = bigger;
      bigger = 2 * root;
   }
   num[root] = key;
} //koniec siftDown
```

Teraz możemy już napisać kod metody `heapSort`; oto on.

```
public static void heapSort(int[] num, int n) {
//sortujemy zakres tablicy od num[1] do num[n]
//przekształcamy tablicę w kopiec
```


Kup ksi k   Pole ksi k

```
   for (int k = n / 2; k >= 1; k--) siftDown(num[k], num, k, n);

   for (int k = n; k > 1; k--) {
      int item = num[k]; //pobieramy aktualnie ostatni element
      num[k] = num[1];   //przenosimy wierzchołek kopca do ostatniego elementu
      siftDown(item, num, 1, k-1); //odtwarzamy warunek kopca w zakresie od 1 do k–1
   }
} //koniec heapSort
```

Działanie metody `heapSort` możemy sprawdzić przy użyciu programu P9.1.

Program P9.1

```
import java.io.*;
public class HeapSortTest {
   public static void main(String[] args) throws IOException {
      int[] num = {0, 37, 25, 43, 65, 48, 84, 73, 18, 79, 56, 69, 32};
      int n = 12;
      heapSort(num, n);
      for (int h = 1; h <= n; h++) System.out.printf("%d ", num[h]);
      System.out.printf("\n");
   }

   public static void heapSort(int[] num, int n) {
   //sortujemy zakres tablicy od num[1] do num[n]
   //przekształcamy tablicę w kopiec
      for (int k = n / 2; k >= 1; k--) siftDown(num[k], num, k, n);

      for (int k = n; k > 1; k--) {
         int item = num[k]; //pobieramy aktualnie ostatni element
         num[k] = num[1];   //przenosimy wierzchołek kopca do ostatniego elementu
         siftDown(item, num, 1, k-1); //odtwarzamy warunek kopca w zakresie od 1 do k–1
      }
   } //koniec heapSort

   public static void siftDown(int key, int[] num, int root, int last) {
      int bigger = 2 * root;
      while (bigger <= last) { //dopóki jest co najmniej jedno dziecko
         if (bigger < last) //istnieje także prawe dziecko; znajdujemy większe
            if (num[bigger+1] > num[bigger]) bigger++;
         //'bigger' zawiera indeks większego dziecka
         if (key >= num[bigger]) break;
         //wartość key jest mniejsza; wybieramy num[bigger]
         num[root] = num[bigger];
         root = bigger;
         bigger = 2 * root;
      }
      num[root] = key;
   } //koniec siftDown

} //koniec klasy HeapSortTest
```

Po uruchomieniu program P9.1 wygeneruje następujące wyniki (elementy `num[1]` do `num[12]` zostaną posortowane).

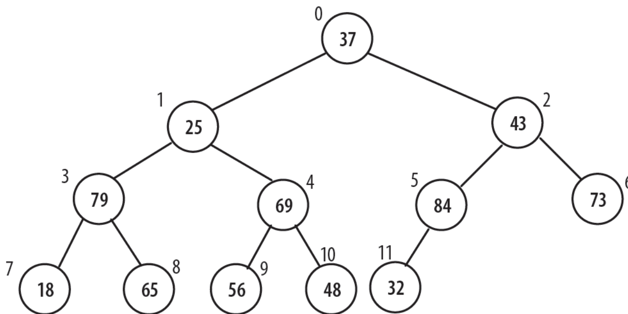
```
18 25 32 37 43 48 56 65 69 73 79 84
```


Kup ksi k   Pole ksi k

Uwaga programistyczna: w przedstawionej postaci metoda `heapSort` sortuje tablicę, przy założeniu, że *n* elementów zostało zapisanych w niej, zaczynając od indeksu `1` do `n`. Gdyby wartości miały być zapisywane w komórkach o indeksach od `0` do `n-1`, konieczne byłoby wprowadzenie stosownych zmian, odpowiadających następującym obserwacjom.

- Korzeniem drzewa jest element `num[0]`.
- Lewym dzieckiem wierzchołka `h` jest wierzchołek `2h+1`, jeśli `2h+1` < `n`.
- Prawym dzieckiem wierzchołka `h` jest wierzchołek `2h+2`, jeśli `2h+2` < `n`.
- Rodzicem wierzchołka `h` jest wierzchołek `(h-1)/2` (przy czym jest to dzielenie całkowite).
- Ostatnim wierzchołkiem, który nie jest liściem, jest wierzchołek `(n-2)/2` (przy czym jest to dzielenie całkowite).

Wszystkie te obserwacje można zweryfikować, analizując drzewo (`n` = 12) przedstawione na rysunku 9.6.



Rysunek 9.6. *Drzewo binarne zapisane w tablicy zaczyna się od elementu o indeksie 0*

Zachęcamy do napisania metody `heapSort` w taki sposób, by sortowała tablicę w zakresie `num[0..n-1]`. W ramach podpowiedzi należy zwrócić uwagę, że jedyną zmianą, jaką trzeba w tym celu wprowadzić w kodzie metody `siftDown`, jest sposób obliczania wartości zmiennej `bigger` — zamiast `2 * root` należy użyć wyrażenia `2 * root + 1`.

9.2. Budowanie kopca przy użyciu metody siftUp

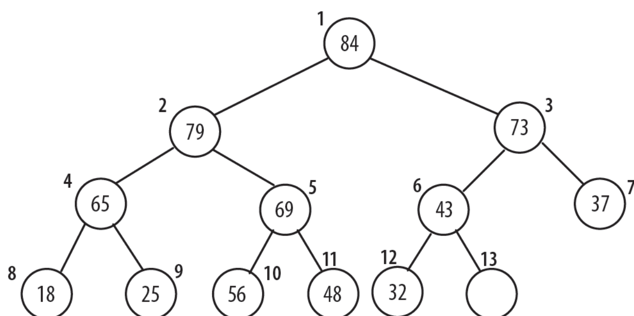
Przeanalizujmy problem dodawania nowego wierzchołka do istniejącego kopca. Konkretnie rzecz biorąc, załóżmy, że elementy tablicy `num[1]` do `num[n]` zawierają kopiec. Chcemy dodać do niego nową liczbę `newKey`, w taki sposób, by `num[1]` do `num[n+1]` tworzyły kopiec zawierający wartość `newKey`. Zakładamy przy tym, że tablica zawierająca kopiec jest na tyle duża, by można w niej umieścić nowy klucz.

Załóżmy np., że dysponujemy kopcem przedstawionym na rysunku 9.7 i chcemy dodać do niego liczbę `40`. Po dodaniu kolejnej liczby tablica będzie zawierać 13 elementów. Przyjmujemy, że liczba `40` została początkowo umieszczona w komórce `num[13]` (jednak na razie jeszcze jej nie zapisujemy w komórce) i porównujemy ją z rodzicem, czyli liczbą `43` umieszczoną w komórce `num[6]`. Ponieważ `40` jest mniejsze od `43`, zatem warunek kopca jest spełniony, a my możemy umieścić nową liczbę w komórce `num[13]`, co zakończy proces dodawania.

Załóżmy jednak, że chcemy dodać do kopca liczbę `80`. Ponownie wyobrażamy sobie, że umieszczamy ją w komórce `num[13]` (choć jeszcze tego nie robimy) i porównujemy z rodzicem, czyli komórką `num[6]` zawierającą wartość `43`. Ponieważ `80` jest większe od `43`, zatem przenosimy `43` do `num[13]` i wyobrażamy sobie, że zapisujemy liczbę `80` w komórce `num[6]`.

Następnie porównujemy `80` z wartością nowego rodzica — wartością komórki `num[3]` — czyli z liczbą `73`. Ponieważ `80` jest większe, zatem przenosimy `43` do komórki `num[6]` i wyobrażamy sobie, że zapisujemy `80` w komórce `num[3]`.

Kup ksi k Pole ksi k

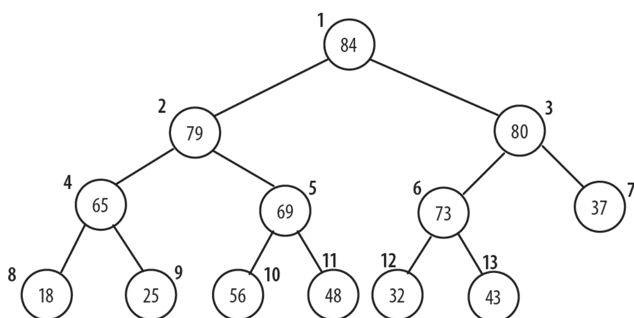


Rysunek 9.7. *Kopiec, do którego dodamy nowy element*

W końcu porównujemy `80` z wartością nowego rodzica — wartością komórki `num[1]` — czyli z liczbą `84`. Ponieważ liczba `80` jest mniejsza, zatem zapisujemy ją w `num[3]` i przetwarzanie zostaje zakończone.

Należy zwrócić uwagę, że gdybyśmy do kopca dodawali liczbę 90, wartość `84` zostałaby przeniesiona do komórki `num[3]`, a 90 została zapisana w komórce `num[1]`. W ten sposób nowy element stanie się największą wartością kopca.

Na rysunku 9.8 przedstawiono kopiec po dodaniu do niego liczby 80.



Rysunek 9.8. *Kopiec po dodaniu liczby 80*

Poniższy kod pozwala dodać wartość `newKey` do kopca zapisanego w tablicy `num`, w zakresie `num[1]` do `num[n]`.

```
child = n + 1;
parent = child / 2;
while (parent > 0) {
   if (newKey <= num[parent]) break;
   num[child] = num[parent]; //przenosimy rodzica w dół
   child = parent;
   parent = child / 2;
}
num[child] = newKey;
n = n + 1;
```

Opisany powyżej proces jest zazwyczaj określany jako **przesiewanie w górę** (ang. *sifting up*). Możemy przepisać go w formie metody `siftUp`. Zakładamy, że do metody będzie przekazywana tablica `heap[1..n]`, taka że `heap[1..n-1]` zawiera kopiec, a `heap[n]` zawiera element do „przesiania w górę”. W efekcie metoda ma zapewnić, że tablica `heap[1..n]` będzie zawierać kopiec. Innymi słowy, element `heap[n]` pełni rolę zmiennej `newKey` z powyższych rozważań.

Kod metody `siftUp` przedstawimy jako fragment programu P9.2, który tworzy kopiec na podstawie liczb odczytywanych z pliku *heap.in*.

Kup ksi k Pole ksi k

Program P9.2

```
import java.io.*;
import java.util.*;
public class SiftUpTest {
   final static int MaxHeapSize = 100;
   public static void main (String[] args) throws IOException {
      Scanner in = new Scanner(new FileReader("heap.in"));
      int[] num = new int[MaxHeapSize + 1];
      int n = 0, number;

      while (in.hasNextInt()) {
         number = in.nextInt();
         if (n < MaxHeapSize) {  //sprawdzamy, czy tablica jest dostatecznie duża
            num[++n] = number;
            siftUp(num, n);
         }
      }

      for (int h = 1; h <= n; h++) System.out.printf("%d ", num[h]);
      System.out.printf("\n");
      in.close();
   } //koniec main

   public static void siftUp(int[] heap, int n) {
    //heap[1] do heap[n−1] zawiera kopiec
    //przesiewamy wartość heap[n] w górę kopca, tak by heap[1..n] zawierała kopiec
      int siftItem = heap[n];
      int child = n;
      int parent = child / 2;
      while (parent > 0) {
         if (siftItem <= heap[parent]) break;
         heap[child] = heap[parent]; //przenosimy rodzica w dół
         child = parent;
         parent = child / 2;
      }
      heap[child] = siftItem;
   } //koniec siftUp

} //koniec klasy SiftUpTest
```

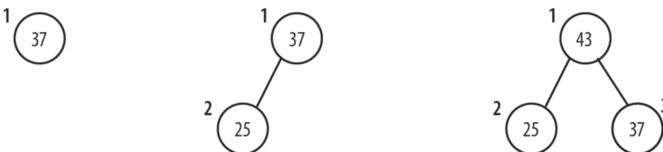
Załóżmy, że plik *heap.in* zawiera następujące liczby:

```
37 25 43 65 48 84 73 18 79 56 69 32
```

Program P9.2 zbuduje kopiec (opisany poniżej) i wyświetli następujące wyniki.

```
84 79 73 48 69 37 65 18 25 43 56 32
```

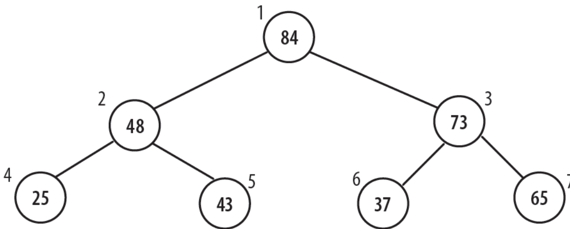
Po wczytaniu liczb 37, 25 oraz 43 kopiec będzie miał postać przedstawioną na rysunku 9.9.



Rysunek 9.9. *Kopiec po przetworzeniu liczb 37, 25 i 43*

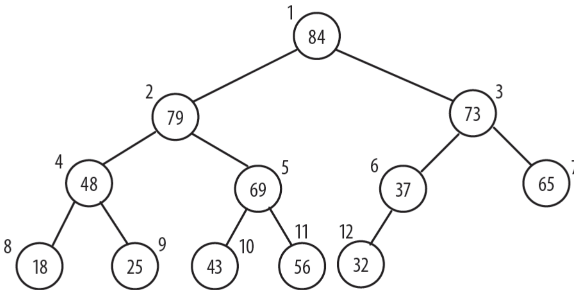
Kup ksi k   Pole ksi k

Po wczytaniu kolejnych liczb, takich jak 65, 48, 84 i 73, kopiec przyjmie postać przedstawioną na rysunku 9.10.



Rysunek 9.10. *Kopiec po przetworzeniu liczb 65, 48, 84 i 73*

Po wczytaniu kolejnych liczb, 18, 79, 56, 69 i 32, kopiec przyjmie postać przedstawioną na rysunku 9.11.



Rysunek 9.11. *Ostateczna postać kopca po przetworzeniu liczb 18, 79, 56, 69 i 32*

Warto zwrócić uwagę, że kopiec z rysunku 9.11 jest inny niż przedstawiony na rysunku 9.3, choć oba zostały utworzone na podstawie tych samych liczb. Nie zmieniło się jednak to, że największa z liczb umieszczonych w kopcu, czyli `84`, znajduje się w jego korzeniu.

Jeśli wartości zostały już zapisane w tablicy `num[1..n]`, możemy przekształcić je w kopiec, używając następującego wywołania:

```
for (int k = 2; k <= n; k++) siftUp(num, k);
```

9.3. Analiza algorytmu sortowania przez kopcowanie

Która z metod, `siftUp` czy `siftDown`, lepiej nadaje się do tworzenia kopca? Trzeba pamiętać, że w większości przypadków liczba przesunięć wierzchołka wyniesie $\log_2 n$.

W metodzie `siftDown` przetwarzamy $n/2$ wierzchołków i w ramach każdego etapu wykonujemy dwa porównania: jedno, by znaleźć większe dziecko, i drugie, by porównać je z wartością wierzchołka. Uproszczona analiza pokazuje, że w najgorszym przypadku będziemy musieli wykonać $2 * n/2 * \log_2 n = n\log_2 n$ porównań. Jednak nieco bardziej dokładna analiza pozwala wykazać, że konieczne będzie wykonanie co najwyżej $4n$ porównań.

W metodzie `siftUp` przetwarzamy $n-1$ wierzchołków. W każdym z etapów wykonujemy jedno porównanie: wierzchołka z jego rodzicem. Uproszczona analiza pokazuje, że w najgorszym przypadku wykonamy $(n-1)\log_2 n$ porównań. Może się jednak zdarzyć, że wszystkie węzły będą musiały przebyć całą drogę, aż do korzenia drzewa. W takim przypadku mamy $n/2$ wierzchołków, które muszą przebyć drogę o długości $\log_2 n$, co daje łączną liczbę $(n/2)\log_2 n$ porównań. A powyższe rozważania dotyczą wyłącznie liści. Ostatecznie szczegółowa analiza pokazuje, że łączna liczba porównań wykonywanych przez metodę `siftUp` wynosi w przybliżeniu $n\log_2 n$.

Kup ksi k   Pole ksi k

Ta różnica w wydajności jest związana z faktem, że w metodzie `siftDown` nie trzeba wykonywać żadnych operacji dla połowy wierzchołków (liści), natomiast metoda `siftUp` właśnie dla tych wierzchołków musi wykonać większość operacji.

Niezależnie od tego, której metody użyjemy do utworzenia początkowego kopca, algorytm sortowania przez kopcowanie posortuje tablicę o wielkości n, wykonując przy tym co najwyżej $2n\log_2 n$ porównań oraz $n\log_2 n$ operacji przypisania. To bardzo szybki algorytm. Co więcej, jest to algorytm *stabilny*, w tym znaczeniu, że jego najgorsza wydajność zawsze wynosi $2n\log_2 n$, niezależnie od początkowej kolejności elementów w tablicy.

Aby unaocznić, jak szybkie jest sortowanie przez kopcowanie (oraz wszystkie inne algorytmy sortowania o złożoności rzędu $O(n\log_2 n)$, takie jak sortowanie szybkie lub sortowanie przez scalanie), porównajmy go z algorytmem sortowania przez wybieranie, który podczas sortowania tablicy n-elementowej wykonuje około $1/2n^2$ porównań. Wyniki tego porównania przedstawiono w tabeli 9.1.

Tabela 9.1. *Porównanie sortowania przez kopcowanie z sortowaniem przez wybieranie*

n	wybieranie (porówn.)	kopcowanie (porówn.)	wybieranie (w sekundach)	kopcowanie (w sekundach)
100	5000	1329	0,005	0,001
1000	500 000	19 932	0,5	0,020
10 000	50 000 000	265 754	50	0,266
100 000	5 000 000 000	3 321 928	5000	3,322
1 000 000	500 000 000 000	39 863 137	500 000	39,863

W drugiej oraz trzeciej kolumnie pokazano liczbę porównań, które należy wykonać. Z kolei w ostatnich dwóch kolumnach przedstawiono czasy wykonania każdej z metod (wyrażone w sekundach), przy założeniu, że komputer jest w stanie wykonać milion porównań na sekundę. Aby np. posortować milion elementów, sortowanie przez wybieranie będzie potrzebować 500 tysięcy sekund (prawie 6 dni!), natomiast sortowanie przez kopcowanie poradzi sobie z tym w ciągu niespełna 40 sekund.

9.4. Kopce i kolejki priorytetowe

Kolejka priorytetowa to kolejka, w której poszczególnym elementom są przypisywane pewne „priorytety" określające położenie danego elementu w kolejce. Element z najwyższym priorytetem jest umieszczany na początku kolejki. Poniżej przedstawiono kilka typowych operacji wykonywanych na kolejkach priorytetowych.

- Usunięcie (udostępnienie) elementu o najwyższym priorytecie.
- Dodanie elementu o podanym priorytecie.
- Usunięcie (usunięcie bez udostępniania) elementu z kolejki.
- Zmiana priorytetu elementu i aktualizacja jego położenia zgodnie z nowym priorytetem.

Priorytet możemy sobie wyobrazić jako liczbę całkowitą — im wyższa liczba, tym wyższy priorytet.

Od razu możemy zgadnąć, że jeśli zaimplementujemy taką kolejkę jako kopiec maksymalny, element o najwyższym priorytecie znajdzie się w jego korzeniu, dzięki czemu bardzo łatwo będzie można go usunąć. Reorganizacja kopca będzie wymagać „przesiania w dół" ostatniego elementu z korzenia kopca.

Dodanie nowego elementu będzie wymagać umieszczenia go za aktualnie ostatnim elementem kopca i przesiania w górę, aż do momentu określenia właściwego położenia.

Aby usunąć z kolejki dowolny element, konieczna będzie znajomość jego położenia. Sam proces usunięcia będzie polegał na zamienieniu usuwanego elementu z aktualnie ostatnim elementem kopca, a następnie przesianiu go w górę, aż do określenia właściwego położenia. W efekcie kopiec zmniejszy się o jeden element.

Kup ksi k Pole ksi k

Jeśli zmienimy priorytet elementu, może się okazać, że konieczne jest przesianie go w górę lub w dół, w celu umieszczenia w odpowiednim położeniu. Oczywiście, w zależności od zmiany, może się okazać, że element pozostanie w swoim oryginalnym położeniu.

W wielu sytuacjach (np. w kolejkach zadań stosowanych w systemach wielozadaniowych) priorytet zadania może się zwiększać z upływem czasu, dzięki czemu w końcu zostanie ono wykonane. W takich sytuacjach po każdej zmianie priorytetu zadanie jest przesuwane coraz bliżej korzenia kopca; jak można się domyślić, wymaga to jedynie przesiania elementów w górę.

W typowych rozwiązaniach informacje o elementach kolejki priorytetowej są przechowywane w innej strukturze danych, w której można je łatwo odnaleźć, np. w drzewie binarnym. Jedno pole wierzchołka będzie zawierać indeks elementu tablicy używanej do przechowywania kolejki priorytetowej.

Kontynuując przykład priorytetowej kolejki zadań, załóżmy, że chcemy dodać do niej nowy element. Możemy przeszukać drzewo na podstawie np. numeru zadania i dodać dany element do drzewa. Wartość jego priorytetu posłuży do określenia miejsca w kolejce, w którym zadanie będzie umieszczone. Położenie to zostanie następnie zapisane w wierzchołku drzewa.

Jeśli później priorytet zadania ulegnie zmianie, zmienione zostanie także położenie zadania w kolejce, a jego nowe położenie ponownie będzie zapisane w wierzchołku drzewa. Warto zwrócić uwagę, że zmiana położenia tego elementu może także powodować zmianę położenia innych elementów w kolejce (które będą przesuwane w górę lub w dół kopca), zatem także dla tych elementów konieczne będzie przeprowadzenie aktualizacji drzewa.

9.5. Sortowanie listy elementów przy użyciu sortowania szybkiego

U podstaw algorytmu sortowania szybkiego (ang. *quicksort*) leży idea *podziału listy* na dwie części, względem pewnego, wybranego elementu, nazywanego **elementem rozdzielającym** (ang. *pivot*). Załóżmy np., że naszym zadaniem jest posortowanie następującej tablicy liczb.

num

53	12	98	63	18	32	80	46	72	21
1	2	3	4	5	6	7	8	9	10

Taką tablicę możemy *podzielić* względem pierwszej wartości, 53. Oznacza to umieszczenie wartości 53 w takim miejscu, że wszystkie wartości znajdujące się w tablicy na lewo od niej będą mniejsze, a znajdujące się na prawo będą od niej większe lub jej równe. Innymi słowy, algorytm podziału tablicy `num` opiszemy w następujący sposób.

num

21	12	18	32	46	53	80	98	72	63
1	2	3	4	5	6	7	8	9	10

Wartość 53 służy jako *element rozdzielający*. Zostaje umieszczona w komórce o numerze 6. Wszystkie wartości na lewo od 53 są od niej mniejsze, a wszystkie wartości na prawo są od niej większe. Miejsce, w którym jest umieszczony element rozdzielający, nosi nazwę **punktu podziału** (ang. *division point*; oznaczymy je jako `dp`). Z definicji wartość 53 znajduje się już w swoim docelowym, posortowanym położeniu.

Jeśli będziemy w stanie posortować fragmenty tablicy `num[1..dp-1]` oraz `num[dp+1..n]`, to będziemy mogli posortować ją całą. Jednak do wykonania tego sortowania możemy zastosować ten sam proces, a to oznacza, że sortowanie można zaimplementować w formie rozwiązania rekurencyjnego.

Zakładając, że dysponujemy funkcją `partition`, która dzieli podany fragment tablicy i zwraca jego punkt podziału, funkcję `quicksort` możemy napisać w następujący sposób.

Kup ksi k   Pole ksi k

```
public static void quicksort(int[] A, int lo, int hi) {
//sortuje A[lo] do A[hi] w kolejności rosnącej
   if (lo < hi) {
      int dp = partition(A, lo, hi);
      quicksort(A, lo, dp-1);
      quicksort(A, dp+1, hi);
   }
} //koniec quicksort
```

Wywołanie `quicksort(num, 1, n)` posortuje tablicę `num[1..n]` w kolejności rosnącej.

Teraz przyjrzymy się, jak można napisać funkcję `partition`. Załóżmy, że dysponujemy następującą tablicą:

num

53	12	98	63	18	32	80	46	72	21
1	2	3	4	5	6	7	8	9	10

Spróbujemy podzielić ją względem wartości `num[1]`, czyli liczby 53 (elementu rozdzielającego), przechodząc przy tym tablicę tylko jeden raz. Kolejno odczytamy wartości poszczególnych elementów tablicy. Jeśli dany element będzie większy do elementu rozdzielającego, nie robimy nic. Jeśli będzie mniejszy, przeniesiemy go na lewą stronę tablicy. Początkowo zmiennej `lastSmall` przypisujemy wartość 1; w trakcie dalszego działania metody zmienna ta będzie przechowywać indeks ostatniego znanego elementu tablicy, który jest mniejszy od elementu rozdzielającego. Proces podziału tablicy `num` przebiega w następujący sposób.

1. Porównujemy 12 z 53, ponieważ jest mniejsze, dodajemy 1 do wartości `lastSmall` (czyli zmienna ta przyjmie wartość 2) i zamieniamy element `num[2]` z nim samym.
2. Porównujemy 98 z 53; ponieważ 98 jest większe, przechodzimy do następnego elementu.
3. Porównujemy 63 z 53; ponieważ 63 jest większe, przechodzimy do następnego elementu.
4. Porównujemy 18 z 53; ponieważ 18 jest mniejsze, dodajemy 1 do `lastSmall` (zmienna ta będzie mieć aktualnie wartość 3) i zamieniamy `num[3]` (czyli 98) z 18.

   Na tym etapie tablica ma postać:

   num

53	12	18	63	98	32	80	46	72	21
1	2	3	4	5	6	7	8	9	10

5. Porównujemy 32 z 53; ponieważ 32 jest mniejsze, dodajemy 1 do `lastSmall` (zmienna ta będzie mieć aktualnie wartość 4) i zamieniamy `num[4]` (czyli 63) z 32.
6. Porównujemy 80 z 53; ponieważ 80 jest większe, przechodzimy do następnego elementu.
7. Porównujemy 46 z 53; ponieważ 46 jest mniejsze, dodajemy 1 do `lastSmall` (zmienna ta będzie mieć aktualnie wartość 5) i zamieniamy `num[5]` (czyli 98) z 46.

   Na tym etapie tablica ma postać:

   num

53	12	18	32	46	63	80	98	72	21
1	2	3	4	5	6	7	8	9	10

8. Porównujemy 72 z 53; ponieważ 72 jest większe, przechodzimy do następnego elementu.
9. Porównujemy 21 z 53; ponieważ 21 jest mniejsze, dodajemy 1 do `lastSmall` (zmienna ta będzie mieć aktualnie wartość 6) i zamieniamy `num[6]` (czyli 63) z 21.
10. Dotarliśmy do końca tablicy; zamieniamy `num[1]` z `num[lastSmall]`; przenosimy tym samym element rozdzielający w jego docelowe, posortowane położenie (w naszym przykładzie będzie to komórka o numerze 6).

Kup ksi k

Pole ksi k

W efekcie uzyskujemy tablicę o następującej zawartości.

num

21	12	18	32	46	53	80	98	72	63
1	2	3	4	5	6	7	8	9	10

Punkt podziału jest wskazywany przez wartość zmiennej `lastSmall` (czyli 6).

Metodę działającą zgodnie z powyższym opisem zaimplementujemy jako funkcję `partition1`. Jej kod przedstawiono jako fragment programu P9.3, którego można użyć do przetestowania działania obu zaprezentowanych wcześniej metod, czyli `quicksort` oraz `partition1`.

Program P9.3

```
import java.io.*;
public class QuicksortTest {

   public static void main(String[] args) throws IOException {
      int[] num = {0, 37, 25, 43, 65, 48, 84, 73, 18, 79, 56, 69, 32};
      int n = 12;
      quicksort(num, 1, n);
      for (int h = 1; h <= n; h++) System.out.printf("%d ", num[h]);
      System.out.printf("\n");
   }

   public static void quicksort(int[] A, int lo, int hi) {
   //sortuje A[lo] do A[hi] w kolejności rosnącej
      if (lo < hi) {
         int dp = partition1(A, lo, hi);
         quicksort(A, lo, dp-1);
         quicksort(A, dp+1, hi);
      }
   } //koniec quicksort

   public static int partition1(int[] A, int lo, int hi) {
   //dzieli A[lo] do A[hi], używając A[lo] jako elementu rozdzielającego
      int pivot = A[lo];
      int lastSmall = lo;
      for (int j = lo + 1; j <= hi; j++)
         if (A[j] < pivot) {
            ++lastSmall;
            swap(A, lastSmall, j);
         }
      //koniec for
      swap(A, lo, lastSmall);
      return lastSmall;   //zwracamy punkt podziału
   } //koniec partition1

   public static void swap(int[] list, int i, int j) {
   //funkcja zamienia elementy list[i] oraz list[j]
      int hold = list[i];
      list[i] = list[j];
      list[j] = hold;
   }

} //koniec klasy QuicksortTest
```


Kup ksi k Pole ksi k

Po wykonaniu program P9.3 wyświetli następujące wyniki (pokazujące, że elementy tablicy od `num[1]` do `num[12]` zostały posortowane).

```
18 25 32 37 43 48 56 65 69 73 79 84
```

Sortowanie szybkie jest jednym z tych algorytmów sortowania, których wydajność może się wahać w granicach od bardzo dużej do bardzo małej. Zazwyczaj algorytm ten ma złożoność rzędu $O(n\log_2 n)$, a dla danych losowych liczba porównań waha się w granicach pomiędzy $n\log_2 n$ a $3n\log_2 n$. Jednak może ona być znacznie większa.

Idea działania tego algorytmu polega na podzieleniu sortowanego fragmentu na dwie stosunkowo równe części. To, czy uda się to zrobić, czy nie, w znacznej mierze zależy od tego, jaka wartość zostanie wybrana na element rozdzielający.

W przedstawionej funkcji jako element rozdzielający wybieramy pierwszy element sortowanego zakresu. Takie rozwiązanie sprawdzi się w większości przypadków, a zwłaszcza podczas sortowania danych losowych. Jeśli jednak pierwszy element okaże się najmniejszym elementem sortowanego zakresu, cała operacja podziału stanie się bezużyteczna, gdyż element rozdzielający znajdzie się na jego pierwszym miejscu. „Lewa" część zakresu będzie pusta, a „prawa" będzie tylko o jeden element mniejsza od aktualnie sortowanego zakresu. Podobnie stanie się, gdy elementem rozdzielającym okaże się największy element sortowanego zakresu.

Choć nawet w takich przypadkach algorytm sortowania szybkiego spełni swoje zadanie, to jednak będzie działał znacząco wolniej. Jeśli np. tablica będzie już posortowana, sortowanie szybkie będzie działać równie wolno jak sortowanie przez wybieranie.

Jednym z rozwiązań tego problemu jest wybranie jako elementu rozdzielającego losowego elementu tablicy, a nie pierwszego. Choć także w tym przypadku istnieje możliwość trafienia na element najmniejszy (lub największy), jednak stanie się to wyłącznie przez przypadek.

Jeszcze innym rozwiązaniem jest użycie jako elementu rozdzielającego mediany pierwszego (`A[lo]`), ostatniego (`A[hi]`) oraz środkowego (`A[(lo+hi)/2]`) elementu zakresu.

Sugerujemy, żeby spróbować przetestowania różnych sposobów wyboru elementu rozdzielającego.

Przeprowadzone eksperymenty pokazały, że wybór losowego elementu tablicy jako elementu rozdzielającego był szybki i dawał dobre efekty nawet w przypadku sortowania już posortowanych danych. W rzeczywistości, w wielu przypadkach takie rozwiązanie będzie działać szybciej na danych posortowanych niż na danych losowych, co w przypadku algorytmu sortowania szybkiego jest niezwykłym wynikiem.

Jedną z możliwych wad algorytmu sortowania szybkiego jest to, że zależnie od sortowanych danych narzuty związane z wykonywaniem wywołań rekurencyjnych mogą być wysokie. W punkcie 9.5.2 pokazano, jak można zminimalizować to zagrożenie. Z drugiej strony, wielką zaletą tego algorytmu jest niewielkie wykorzystanie dodatkowej pamięci. Warto to porównać z algorytmem `mergesort` (sortowaniem przez scalanie, także rekurencyjnym), wymagającym znacznie więcej dodatkowego miejsca (dokładnie tyle samo, ile wynosi wielkość sortowanej tablicy), którego używa do scalenia sortowanych fragmentów. Żadnej z tych wad nie ma natomiast algorytm sortowania przez kopcowanie. Nie jest to algorytm rekurencyjny i wymaga bardzo niewiele dodatkowej pamięci. Poza tym, zgodnie z informacjami podanymi w podrozdziale 9.3, sortowanie przez kopcowanie jest *stabilne*, pod tym względem, że jego wydajność w najgorszym razie wynosi $2n\log_2 n$ i to niezależnie od porządku elementów w sortowanej tablicy.

9.5.1. Inny sposób podziału

Cel podziału sortowanego zakresu tablicy — czyli utworzenie dwóch części, takich że wszystkie elementy w lewej będą mniejsze od wszystkich elementów w prawej — można uzyskać na wiele sposobów. Pierwsza metoda, przedstawiona wcześniej, umieszczała element rozdzielający w jego docelowym położeniu. Dla odmiany przeanalizujemy teraz nieco inny sposób podziału. Choć także on przeprowadza podział względem elementu rozdzielającego, to jednak *nie umieszcza* go w docelowym położeniu. Jak się jednak przekonamy, nie będzie to żadnym problemem.

Kup ksi k   Pole ksi k

Ponownie załóżmy, że dysponujemy tablicą `num[1..n]`, gdzie `n` = 10.

num

53	12	98	63	18	32	80	46	72	21
1	2	3	4	5	6	7	8	9	10

Jako element rozdzielający wybieramy element 53. Ogólna idea polega na tym, by przeglądać tablicę, zaczynając od prawej, i szukać w niej klucza o wartości mniejszej lub równej wartości elementu rozdzielającego. Następnie tablica jest przeglądana od lewej, w poszukiwaniu klucza, który jest większy lub równy wartości elementu rozdzielającego. W końcu obie te wartości są zamieniane miejscami. Proces ten w efekcie powoduje, że wartości mniejsze są umieszczane z lewej, a większe z prawej strony sortowanego fragmentu tablicy.

Zastosujemy dwie zmienne, `lo` oraz `hi`, które będą oznaczały położenie z lewej oraz z prawej. Początkowo zmiennej `lo` przypisujemy wartość 0, a zmiennej `hi` wartość 11 (`n+1`). Następnie powtarzamy następujące czynności.

1. Odejmujemy 1 od `hi` (czyli zmienna ta przyjmuje wartość 10).
2. Porównujemy `num[hi]`, czyli 21, z 53; ponieważ 21 jest mniejsze, zatrzymujemy przeszukiwanie tablicy od prawej na `hi` = 10.
3. Dodajemy 1 do `lo` (czyli zmienna ta przyjmuje wartość 1).
4. Porównujemy `num[lo]`, czyli 53, z 53; ponieważ 53 nie jest mniejsze, zatrzymujemy przeszukiwanie tablicy od lewej na `lo` = 1.
5. `lo` (1) jest mniejsze od `hi` (10), czyli zamieniamy `num[lo]` z `num[hi]`.
6. Odejmujemy 1 od `hi` (czyli zmienna ta przyjmuje wartość 9).
7. Porównujemy `num[hi]`, czyli 72, z 53; ponieważ 72 jest większe, zmniejszamy wartość `hi` o 1 (przez co zmienna ta przyjmie wartość 8). Porównujemy `num[hi]`, czyli 46, z 53, ponieważ 46 jest mniejsze, zatrzymujemy przeszukiwanie tablicy od prawej na `hi` = 8.
8. Dodajemy 1 do `lo` (czyli zmienna ta przyjmuje wartość 2).
9. Porównujemy `num[lo]`, czyli 12, z 53; ponieważ 12 jest mniejsze, dodajemy 1 do `lo` (zmienna ta przyjmuje wartość 3). Porównujemy `num[lo]`, czyli 98, z 53; ponieważ 98 jest większe, zatrzymujemy przeszukiwanie tablicy od lewej na `lo` = 3.
10. `lo` (3) jest mniejsze od `hi` (8), czyli zamieniamy `num[lo]` z `num[hi]`.

    Na tym etapie działania zmienna `lo` = 3, zmienna `hi` = 8, a zawartość tablicy `num` ma postać:

    num

21	12	46	63	18	32	80	98	72	53
1	2	3	4	5	6	7	8	9	10

11. Odejmujemy 1 od `hi` (czyli zmienna ta przyjmuje wartość 7).
12. Porównujemy `num[hi]`, czyli 80, z 53; ponieważ 80 jest większe, zmniejszamy wartość `hi` o 1 (przez co zmienna ta przyjmie wartość 6). Porównujemy `num[hi]`, czyli 32, z 53, ponieważ 32 jest mniejsze, zatrzymujemy przeszukiwanie tablicy od prawej na `hi` = 6.
13. Dodajemy 1 do `lo` (czyli zmienna ta przyjmuje wartość 4).
14. Porównujemy `num[lo]`, czyli 63, z 53; ponieważ 63 jest większe, zatrzymujemy przeszukiwanie tablicy od lewej na `lo` = 4.
15. `lo` (4) jest mniejsze od `hi` (6), czyli zamieniamy `num[lo]` z `num[hi]`, uzyskując tablicę o następującej zawartości:

    num

21	12	46	32	18	32	80	98	72	53
1	2	3	4	5	6	7	8	9	10

Kup ksi k Pole ksi k

16. Odejmujemy 1 od hi (czyli zmienna ta przyjmuje wartość 5).
17. Porównujemy num[hi], czyli 18, z 53; ponieważ 18 jest mniejsze, zatrzymujemy przeszukiwanie tablicy od prawej na hi = 5.
18. Dodajemy 1 do lo (czyli zmienna ta przyjmuje wartość 5).
19. Porównujemy num[lo], czyli 18, z 53; ponieważ 18 jest mniejsze, dodajemy 1 do lo (zmienna ta przyjmuje wartość 6). Porównujemy num[lo], czyli 63, z 53; ponieważ 63 jest większe, zatrzymujemy przeszukiwanie tablicy od lewej na lo = 6.
20. lo (6) *nie jest mniejsze* od hi (5), zatem algorytm kończy działanie.

Wartość zmiennej hi spełnia tę właściwość, że wartości num[1..hi] są mniejsze od wartości num[hi+1..n]. W naszym przypadku wartości num[1..5] są mniejsze od wartości num[6..10]. Trzeba zwrócić uwagę, że wartość 53 nie znajduje się w swoim docelowym, posortowanym położeniu. Jednak nie stanowi to większego problemu, gdyż w celu posortowania całej tablicy musimy jeszcze posortować jej fragmenty: num[1..hi] oraz num[hi+1..n].

Tak opisaną procedurę podziału możemy zaimplementować w formie następującej metody partition2.

```
public static int partition2(int[] A, int lo, int hi) {
//funkcja zwraca punkt podziału (dp), taki że A[lo..dp] <= A[dp+1..hi]
   int pivot = A[lo];
   --lo; ++hi;
   while (lo < hi) {
      do --hi; while (A[hi] > pivot);
      do ++lo; while (A[lo] < pivot);
      if (lo < hi) swap(A, lo, hi);
   }
   return hi;
} //koniec partition2
```

W przypadku zastosowania tej metody podziału tablicy na dwie części metodę quicksort2 możemy napisać w następujący sposób.

```
public static void quicksort2(int[] A, int lo, int hi) {
//sortuje A[lo] do A[hi] w kolejności rosnącej
   if (lo < hi) {
      int dp = partition2(A, lo, hi);
      quicksort2(A, lo, dp);
      quicksort2(A, dp+1, hi);
   }
}
```

W metodzie partition2 jako element rozdzielający wybieramy pierwszy element sortowanego zakresu tablicy. Jednak zgodnie z informacjami podanymi wcześniej, wybór dowolnego elementu zapewniłby lepsze wyniki. Taki losowy element można wybrać w następujący sposób:

```
swap(A, lo, random(lo, hi));
int pivot = A[lo];
```

W tym przypadku funkcja random będzie mieć postać:

```
public static int random(int m, int n) {
   //zwraca losową liczbę całkowitą z zakresu do m do n włącznie
   return (int) (Math.random() * (n - m + 1)) + m;
}
```


Kup ksi k Pole ksi k

9.5.2. Nierekurencyjna wersja algorytmu sortowania szybkiego

W przedstawionych wcześniej wersjach metody `quicksort` po dokonaniu podziału sortowanego fragmentu tablicy ta sama metoda była wykonywana rekurencyjnie, w celu posortowania najpierw lewej, a następnie prawej części zakresu. Takie rozwiązanie spełnia się doskonale. Jednak może się zdarzyć, że dla dużych wartości `n` liczba wykonywanych wywołań rekurencyjnych stanie się tak duża, że wystąpi błąd „przepełniania buforu rekurencji" (ang. *recursive stack overflow*).

Z przeprowadzonych eksperymentów wynika, że dzieje się tak dla $n = 7000$, jeśli dane były już wcześniej posortowane, a jako element rozdzielający wybrany został pierwszy element sortowanego zakresu. Jednak algorytm działał bardzo dobrze nawet dla $n = 100\,000$, kiedy elementem rozdzielającym został wybrany losowy element tablicy.

Jeszcze innym rozwiązaniem jest napisanie metody `quicksort` w taki sposób, by nie korzystać w niej z rekurencji. Wymaga ona umieszczenia na stosie tych elementów listy, które jeszcze nie zostały posortowane. Można wykazać, że jeśli podlista jest dzielona na dwie części, to posortowanie najpierw *mniejszej* z nich sprawia, że liczba elementów na stosie zostanie ograniczona niemal do $\log_2 n$.

Załóżmy np., że sortujemy tablicę `A[1..99]`, a pierwszy punkt podziału wypadł w elemencie o numerze 40. Załóżmy dodatkowo, że używamy metody `partition2`, która nie umieszcza elementu rozdzielającego w jego docelowym, posortowanym położeniu. A zatem w celu dokończenia sortowania musimy posortować dwa fragmenty tablicy: `A[1..40]` oraz `A[41..99]`. Umieścimy zatem na stosie parę liczb (41, 99) i zajmiemy się posortowaniem fragmentu tablicy `A[1..40]` (czyli krótszej podlisty).

Załóżmy, że punkt podziału fragmentu `A[1..40]` wypadł w elemencie numer 25. Umieszczamy zatem na stosie (1, 25) i przetwarzamy w pierwszej kolejności fragment `A[26..40]`. Na tym etapie prac na stosie znajdują się dwie podlisty — (41, 99) oraz (1, 25) — którymi jeszcze musimy się zająć. Próba posortowania fragmentu `A[26..40]` spowoduje umieszczenie na stosie kolejnej podlisty itd. W naszej implementacji algorytmu na stosie będziemy także umieszczali krótszą podlistę, jednak natychmiast będzie ona zdejmowana z niego i przetwarzana.

Wspominane wcześniej wyniki gwarantują, że na stosie nigdy nie będzie więcej niż $\log_2 99 = 7$ (po zaokrągleniu w górę) elementów. Nawet w razie sortowania $n = 1\,000\,000$ elementów mamy gwarancję, że liczba elementów umieszczonych na stosie nigdy nie przekroczy 20.

Oczywiście, tym stosem musimy zarządzać samodzielnie. Każdy element stosu będzie zawierał dwie liczby całkowite (nazwiemy je `left` i `right`), określające, że do posortowania pozostaje jeszcze zakres tablicy pomiędzy elementami `left` i `right`. Klasę `NodeData` możemy zdefiniować w następujący sposób.

```
class NodeData {
   int left, right;

   public NodeData(int a, int b) {
      left = a;
      right = b;
   }

   public static NodeData getRogueValue() {return new NodeData(-1, -1);}

} //koniec klasy NodeData
```

W programie zastosujemy implementację stosu z podrozdziału 4.3. Teraz, bazując na przedstawionych wcześniej informacjach, możemy już napisać metodę `quicksort3`. Przedstawiono ją jako fragment programu P9.4. Program ten wczytuje liczby z pliku *quick.in*, sortuje je przy użyciu metody `quicksort3`, a następnie wyświetla posortowane liczby, po dziesięć w jednym wierszu.

Program P9.4

```
import java.io.*;
import java.util.*;
public class Quicksort3Test {
   final static int MaxNumbers = 100;
```


Kup książkę Poleć książkę

```
public static void main (String[] args) throws IOException {
   Scanner in = new Scanner(new FileReader("quick.in"));
   int[] num = new int[MaxNumbers+1];
   int n = 0, number;

   while (in.hasNextInt()) {
      number = in.nextInt();
      if (n < MaxNumbers) num[++n] = number; //zapisujemy, jeśli w tablicy jest miejsce
   }

   quicksort3(num, 1, n);
   for (int h = 1; h <= n; h++) {
      System.out.printf("%d ", num[h]);
      if (h % 10 == 0) System.out.printf("\n"); //wyświetlamy po 10 liczb w wierszu
   }
   System.out.printf("\n");
} //koniec main

public static void quicksort3(int[] A, int lo, int hi) {
   Stack S = new Stack();
   S.push(new NodeData(lo, hi));
   int stackItems = 1, maxStackItems = 1;

   while (!S.empty()) {
      --stackItems;
      NodeData d = S.pop();
      if (d.left < d.right) { //jeśli podlista jest > 1 elementu
         int dp = partition2(A, d.left, d.right);
         if (dp - d.left + 1 < d.right - dp) {  //porównujemy długości podlist
            S.push(new NodeData(dp+1, d.right));
            S.push(new NodeData(d.left, dp));
         }
         else {
            S.push(new NodeData(d.left, dp));
            S.push(new NodeData(dp+1, d.right));
         }
         stackItems += 2;     //dwa elementy dodawane do stosu
      } //koniec if
      if (stackItems > maxStackItems) maxStackItems = stackItems;
   } //koniec while
   System.out.printf("Maksymalna liczba elementów na stosie: %d\n\n", maxStackItems);
} //koniec quicksort3

public static int partition2(int[] A, int lo, int hi) {
//funkcja zwraca punkt podziału (dp), taki że A[lo..dp] <= A[dp+1..hi]
   int pivot = A[lo];
   --lo; ++hi;
   while (lo < hi) {
      do --hi; while (A[hi] > pivot);
      do ++lo; while (A[lo] < pivot);
      if (lo < hi) swap(A, lo, hi);
   }
   return hi;
} //koniec partition2

public static void swap(int[] list, int i, int j) {
//funkcja zamienia elementy list[i] oraz list[j]
```

```
      int hold = list[i];
      list[i] = list[j];
      list[j] = hold;
   } //koniec swap

} //koniec klasy Quicksort3Test

class NodeData {
   int left, right;

   public NodeData(int a, int b) {
      left = a;
      right = b;
   }

   public static NodeData getRogueValue() {return new NodeData(-1, -1);}

} //koniec klasy NodeData

class Node {
   NodeData data;
   Node next;

   public Node(NodeData d) {
      data = d;
      next = null;
   }
} //koniec klasy Node

class Stack {
   Node top = null;

   public boolean empty() {
      return top == null;
   }

   public void push(NodeData nd) {
      Node p = new Node(nd);
      p.next = top;
      top = p;
   } //koniec push

   public NodeData pop() {
      if (this.empty())return NodeData.getRogueValue();
      NodeData hold = top.data;
      top = top.next;
      return hold;
   } //koniec pop

} //koniec klasy Stack
```

W metodzie `quicksort3` po wykonaniu metody `partition2` porównywane są długości obu podlist, po czym na stosie jest umieszczana najpierw dłuższa, a następnie krótsza z nich. W ten sposób krótsza lista zostanie zdjęta ze stosu jako pierwsza i przetworzona przed dłuższą.

Oprócz tego, do metody `quicksort3` dodano także instrukcje, które śledzą maksymalną liczbę elementów umieszczonych na stosie. Kiedy zastosowaliśmy program do posortowania 100 000 liczb, maksymalna liczba elementów umieszczonych na stosie wyniosła 13. To mniej niż wynosi teoretyczne maksimum, $\log_2 100000 = 17$ (po zaokrągleniu w górę).

Kup ksi k Pole ksi k

Załóżmy, że plik *quick.in* zawiera następujące liczby:

```
43 25 66 37 65 48 84 73 60 79 56 69 32 87 23 99 85 28 14 78 39 51 44 35
46 90 26 96 88 31 17 81 42 54 93 38 22 63 40 68 50 86 75 21 77 58 72 19
```

Po uruchomieniu program P9.4 wygeneruje poniższe wyniki.

```
Maksymalna liczba elementów na stosie: 5

14 17 19 21 22 23 25 26 28 31
32 35 37 38 39 40 42 43 44 46
48 50 51 54 56 58 60 63 65 66
68 69 72 73 75 77 78 79 81 84
85 86 87 88 90 93 96 99
```

W przedstawionej postaci, nawet jeśli podlista będzie zawierać jedynie dwa elementy, metoda wykona cały proces sortowania, włącznie z przeprowadzaniem podziału, sprawdzaniem długości podlist i umieszczaniem na stosie dwóch elementów. To całkiem sporo pracy jak na posortowanie dwóch elementów.

Wydajność działania algorytmu sortowania szybkiego można dodatkowo poprawić przy użyciu jakiejś prostszej metody (takiej jak sortowanie przez wstawianie) do sortowania podlist krótszych od pewnej predefiniowanej długości (np. składających się z mniej niż 8 elementów). Warto wprowadzić taką modyfikację w metodzie `quicksort` i sprawdzić efekty dla różnych wartości predefiniowanej długości.

9.5.3. Znajdowanie *k*-tej najmniejszej liczby

Przeanalizujemy teraz problem znajdowania *k*-tej najmniejszej liczby na liście zawierającej *n* liczb. Jednym ze sposobów rozwiązania tego problemu jest posortowanie listy i wybranie *k*-tej wartości. Jeśli liczby będą zapisane w tablicy `A[1..n]`, wystarczy w tym celu pobrać z tablicy element `A[k]`.

Innym, bardziej wydajnym sposobem jest zastosowanie pomysłu podziału na części. Wykorzystamy w tym celu metodę `partition`, która umieszcza element rozdzielający w docelowym, posortowanym miejscu. Załóżmy, że dysponujemy tablicą `A[1..99]` i przyjmijmy, że wywołanie metody `partition` zwróciło punkt podziału o wartości `40`. Oznacza to, że element rozdzielający został umieszczony w komórce `A[40]`, wszystkie mniejsze liczby znajdują się po jego lewej stronie, a większe — po prawej. Innymi słowy, 40. najmniejsza wartość została umieszczona w elemencie tablicy `A[40]`. A zatem, jeśli *k* wynosi 40, od razu uzyskaliśmy rozwiązanie.

A co będzie, jeśli wartość *k* wynosi 59? Wiemy, że 40 najmniejszych liczb zajmuje zakres `A[1..40]`. W takim razie 59. liczba musi się znajdować w zakresie `A[41..99]`, więc możemy ograniczyć nasze poszukiwania do tego obszaru. Innymi słowy, jedno wywołanie metody `partition` pozwoliło wyeliminować 40 liczb. Takie rozwiązanie przypomina nieco *wyszukiwanie binarne*.

Załóżmy następnie, że wywołanie metody `partition` zwróciło wartość 65. Znamy zatem 65. najmniejszą liczbę, a 59. będzie się znajdować w zakresie `A[41..65]`. A zatem z dalszych poszukiwań możemy wykluczyć zakres `A[66..99]`. Taki proces możemy powtarzać, redukując za każdym razem wielkość zakresu tablicy, w którym znajduje się 59. najmniejsza liczba w tablicy. W końcu metoda `partition` zwróci wartość 59 i uzyskamy poszukiwaną odpowiedź.

W poniższym kodzie przedstawiono jedną z możliwych postaci metody `kthSmall`, wykorzystującą metodę `partition1`.

```
public static int kthSmall(int[] A, int k, int lo, int hi) {
//zwraca k-tą najmniejszą liczbę z zakresu od A[lo] do A[hi]
   int kShift = lo + k - 1; //przesuwa k do podanego położenia, A[lo..hi]
   if (kShift < lo || kShift > hi) return -9999;
   int dp = partition1(A, lo, hi);
   while (dp != kShift) {
      if (kShift < dp) hi = dp - 1; //k-ta najmniejsza liczba znajduje się w lewej części
      else lo = dp + 1; //k-ta najmniejsza liczba znajduje się w prawej części
```

```
        dp = partition1(A, lo, hi);
    }
    return A[dp];
} //koniec kthSmall
```

Przykładowo wywołanie `kthSmall(num, 59, 1, 99)` zwróci 59. najmniejszą liczbę z zakresu `num[1..99]`. Warto jednak zauważyć, że wywołanie `kthSmall(num, 10, 30, 75)` zwróci 10. najmniejszą liczbę z zakresu `num[30..75]`.

W ramach ćwiczenia warto napisać rekurencyjną wersję metody `kthSmall`.

9.6. Sortowanie Shella (z użyciem malejących odstępów)

W sortowaniu Shella (nazwanego tak od nazwiska jego twórcy — Donalda Shella) stosuje się sekwencję *odstępów*, które zarządzają procesem sortowania. Algorytm wykonuje kilka przebiegów przez sortowaną tablicę danych, przy czym ostatni z nich sprowadza się do zwyczajnego sortowania przez wstawianie. We wcześniejszych przebiegach sortowane są elementy rozmieszczone w określonych odstępach (np. co pięć elementów), przy czym używana jest ta sama technika sortowania przez wstawianie.

Aby np. posortować poniższą tablicę, zastosujemy trzy odstępy: 8, 3 i 1.

num

67	90	28	84	29	58	25	32	16	64	13	71	82	10	51	57
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Odstępy te są stopniowo zmniejszane (stąd określenie **sortowanie z użyciem malejących odstępów**), aż w końcu przybierają wartość 1.

Zaczynamy do odstępu o wartości 8, a zatem sortujemy co ósmy element tablicy, czyli sortujemy elementy 1. i 9., 2. i 10., 3. i 11., 4. i 12., 5. i 13., 6. i 14., 7. i 15. oraz 8. i 16. Spowoduje to przekształcenie zawartości tablicy `num` do postaci:

num

16	64	13	71	29	10	25	32	67	90	28	84	82	58	51	57
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Następnie użyjemy odstępu o wartości 3, co oznacza, że sortujemy co trzeci jej element. Innymi słowy, sortujemy elementy (1., 4., 7., 10., 13. i 16.), (2., 5., 8., 11. i 14.) oraz (3., 6., 9., 12. i 15.). Uzyskujemy w ten sposób tablicę w postaci:

num

16	28	10	25	29	13	57	32	51	71	58	67	82	64	84	90
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Należy zwrócić uwagę, że po każdym takim kroku tablica jest w coraz większym stopniu posortowana. W końcu przeprowadzamy sortowanie z odstępem o wartości 1, czyli sortujemy całą listę, nadając jej przy tym ostateczną, posortowaną postać.

num

10	13	16	25	28	29	32	51	57	58	64	67	71	82	84	90
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Można by zapytać, dlaczego na samym początku nie wykonaliśmy sortowania z odstępem 1, sortując od razu całą zawartość tablicy? Cała idea tego algorytmu polega na tym, że gdy dochodzimy już do użycia odstępu 1, tablica jest w mniejszym lub większym stopniu posortowana, a jeśli używamy przy tym metody, która lepiej działa na częściowo posortowanych danych (takiej jak sortowanie przez wstawianie), proces sortowania może być wykonany bardzo szybko. Warto sobie przypomnieć, że sortowanie przez wstawianie

Kup ksi k Pole ksi k

może się ograniczyć do wykonania jedynie *n* porównań dla *n* elementów (jeśli dane będą już posortowane) bądź wykonać ich aż $1/2n^2$ (jeśli dane będą posortowane w kolejności malejącej, a my chcemy je posortować w kolejności rosnącej).

Gdy odstępy są duże, podlisty sortowanych liczb są niewielkie. W naszym przypadku przy odstępie o wartości 8 każda sortowana podlista składała się wyłącznie z dwóch elementów. Można oczekiwać, że małe podlisty będą sortowane szybko. Przy mniejszych odstępach liczba elementów w sortowanych podlistach rośnie. Kiedy jednak do tego dojdzie, zawartość tablicy będzie już częściowo posortowana i jeśli wykorzystamy algorytm sortowania, który potrafi wykorzystać takie uporządkowanie elementów, będziemy w stanie posortować je szybko.

W naszym rozwiązaniu do sortowania z odstępem *h*, przy założeniu, że *h* jest większe od 1, zastosujemy nieco zmodyfikowaną wersję sortowania przez wstawianie.

Kiedy w przypadku tego algorytmu dochodzimy do przetworzenia elementu `num[k]`, zakładamy, że elementy `num[1..k-1]` są posortowane i wstawiamy element `num[k]` w taki sposób, by elementy `num[1..k]` były posortowane.

Załóżmy, że odstęp wynosi `h` i zastanówmy się, w jaki sposób możemy przetworzyć element `num[k]`, gdzie `k` jest dowolnym, prawidłowym indeksem tablicy. Pamiętajmy, że naszym celem jest posortowanie podlisty elementów oddalonych od siebie o `h`. A zatem, sortując `num[k]`, musimy sprawdzić elementy `num[k-h]`, `num[k-2h]`, `num[k-3h]` itd., przy założeniu, że wszystkie te elementy leżą w obszarze tablicy. Kiedy zaczynamy przetwarzać element `num[k]` i wcześniejsze elementy, rozmieszczone w odstępach co `h`, są już posortowane, wystarczy wstawić element `num[k]` pomiędzy nie w taki sposób, by cała podlista kończąca się na `num[k]` była posortowana.

Aby zilustrować ten proces, załóżmy, że `h = 3`, a `k = 4`. W tablicy przed `num[4]` znajduje się tylko jeden element — `num[1]`. A zatem, kiedy zaczynamy przetwarzać `num[4]`, możemy założyć, że element `num[1]` jest posortowany. Wstawiamy zatem `num[4]` w takie miejsce, że podlista składająca się z elementów `num[1]` i `num[4]` będzie posortowana.

Podobnie, przed elementem `num[5]` znajduje się tylko jeden element oddalony od niego o 3 i jest to `num[2]`. A zatem, kiedy zaczynamy przetwarzać `num[5]`, możemy założyć, że element `num[2]` jest posortowany. Wstawiamy zatem `num[4]` w takie miejsce, że podlista składająca się z elementów `num[2]` i `num[5]` będzie posortowana. Podobnie postępujemy z elementami `num[3]` i `num[6]`.

Kiedy zaczynamy przetwarzać element `num[7]`, przed nim w tablicy będą się znajdowały dwa, już posortowane elementy — `num[1]` i `num[4]`. Wstawiamy zatem `num[7]` w takie miejsce, że podlista składająca się z elementów `num[1]`, `num[4]` i `num[7]` będzie posortowana.

Kiedy zaczynamy przetwarzać element `num[8]`, przed nim w tablicy będą się znajdowały dwa, już posortowane elementy — `num[2]` i `num[5]`. Wstawiamy zatem `num[8]` w takie miejsce, że podlista składająca się z elementów `num[2]`, `num[5]` i `num[8]` będzie posortowana.

Następnie, kiedy zaczynamy przetwarzać element `num[9]`, przed nim w tablicy będą się znajdowały dwa, już posortowane elementy — `num[3]` i `num[6]`. Wstawiamy zatem `num[9]` w takie miejsce, że podlista składająca się z elementów `num[3]`, `num[6]` i `num[9]` będzie posortowana.

Kiedy zaczynamy przetwarzać element `num[10]`, przed nim w tablicy będą się znajdowały trzy, już posortowane elementy — `num[1]`, `num[4]` oraz `num[7]`. Wstawiamy zatem `num[10]` w takie miejsce, że podlista składająca się z elementów `num[1]`, `num[4]`, `num[7]` i `num[10]` będzie posortowana.

I tak dalej. Zaczynamy od `h+1`, przechodzimy tablicę, przetwarzając każdy jej element i porównując go z wcześniejszymi elementami w tablicy, rozmieszczonymi w odstępach będących wielokrotnością `h`.

W naszym przykładzie, gdy `h = 3`, stwierdziliśmy, że musimy przetworzyć podlisty składające się z elementów (1., 4., 7., 10., 13., 16.), (2., 5., 8., 11., 14.) oraz (3., 6., 9., 12., 15.). To prawda; jednak nasz algorytm nie będzie sortował elementów (1., 4., 7., 10., 13., 16.), a następnie (2., 5., 8., 11., 14.), by zakończyć sortowaniem elementów (3., 6., 9., 12., 15.).

Zamiast tego będzie je sortował równolegle, sortując poszczególne elementy w następującej kolejności: (1., 4.), (2., 5.), (3., 6.), (1., 4., 7.), (2., 5., 8.), (3., 6., 9.), (1., 4., 7., 10.), (2., 5., 8., 11.), (3., 6., 9., 12.), (1., 4., 7., 10., 13.), (2., 5., 8., 11., 14.), (3., 6., 9., 12., 15.), i w końcu (1., 4., 7., 10., 13., 16.). Być może wygląda to na dosyć złożone rozwiązanie, jednak pod względem implementacji jest ono łatwiejsze, gdyż wystarczy przetworzyć tablicę, zaczynając od `h+1`.

Kup ksi k Pole ksi k

Poniższa metoda wykona `h`-sortowanie tablicy `A[1..n]`.

```
public static void hsort(int[] A, int n, int h) {
   for (int k = h + 1; k <= n; k++) {
      int j = k - h;
      int key = A[k];
      while (j > 0 && key < A[j]) {
         A[j + h] = A[j];
         j = j - h;
      }
      A[j + h] = key;
   }
} //koniec hsort
```

Łatwo zauważyć, że gdy `h` przyjmie wartość `1`, powyższa metoda staje się implementacją sortowania przez wstawianie.

Uwaga programistyczna: kiedy chcemy posortować tablicę `A[0..n-1]`, musimy w pętli `while` użyć warunku `j >= 0`, a oprócz tego zmienić instrukcję `for` w następujący sposób:

```
for (int k = h; k < n; k++)
```

Teraz, kiedy dysponujemy sekwencją odstępów $h_t, h_{t-1}, ..., h_1 = 1$, możemy wykonać sortowanie, wywołując metodę `hsort` dla każdego z odstępów, od największego do najmniejszego.

Przedstawiony poniżej program P9.5 wczytuje liczby z pliku *shell.in*, sortuje je przy użyciu sortowania Shella (przy czym używa odstępów 8, 3 i 1), a następnie wyświetla posortowane liczby, umieszczając po 10 w wierszu.

Program 9.5

```
import java.io.*;
import java.util.*;
public class ShellSortTest {
   final static int MaxNumbers = 100;
   public static void main (String[] args) throws IOException {
      Scanner in = new Scanner(new FileReader("shell.in"));
      int[] num = new int[MaxNumbers+1];
      int n = 0, number;
      while (in.hasNextInt()) {
         number = in.nextInt();
         if (n < MaxNumbers) num[++n] = number;
                                          //zapisujemy, jeśli jest miejsce w tablicy
      }

      //wykonujemy sortowanie Shella z przyrostami 8, 3 i 1
      hsort(num, n, 8);
      hsort(num, n, 3);
      hsort(num, n, 1);

      for (int h = 1; h <= n; h++) {
         System.out.printf("%d ", num[h]);
         if (h % 10 == 0) System.out.printf("\n"); //wyświetlamy po 10 liczb w wierszu
      }
      System.out.printf("\n");
   } //koniec main

   public static void hsort(int[] A, int n, int h) {
      for (int k = h + 1; k <= n; k++) {
         int j = k - h;
         int key = A[k];
```


Kup książkę Poleć książkę

```
         while (j > 0 && key < A[j]) {
            A[j + h] = A[j];
            j = j - h;
         }
         A[j + h] = key;
      } //koniec for
   } //koniec hsort

} //koniec klasy ShellSortTest
```

Załóżmy, że plik *shell.in* ma następującą zawartość:

```
43 25 66 37 65 48 84 73 60 79 56 69 32 87 23 99 85 28 14 78 39 51 44 35
46 90 26 96 88 31 17 81 42 54 93 38 22 63 40 68 50 86 75 21 77 58 72 19
```

W takim przypadku program P9.5 wyświetli następujące wyniki.

```
14 17 19 21 22 23 25 26 28 31
32 35 37 38 39 40 42 43 44 46
48 50 51 54 56 58 60 63 65 66
68 69 72 73 75 77 78 79 81 84
85 86 87 88 90 93 96 99
```

Mimochodem można zauważyć, że nasz kod byłby bardziej elastyczny, gdyby wielkości odstępów były zapisane w tablicy (dajmy na to `incr`), a do metody `hsort` były przekazywane kolejne elementy tej tablicy. Załóżmy np., że element `incr[0]` zawiera liczbę odstępów (przykładowo `m`), a kolejne elementy od `incr[1]` do `incr[m]` zawierają wartości kolejnych odstępów, przy czym `incr[m]` = 1. W takim przypadku metodę `hsort` moglibyśmy wywoływać w taki sposób:

```
for (int i = 1; i <= incr[0]; i++) hsort(num, n, incr[i]);
```

Jednym z pytań, na jakie należy znaleźć odpowiedź, jest to, w jaki sposób dobierać wielkość odstępu dla danego *n*. Zaproponowano wiele sposobów rozwiązania tego zagadnienia, a poniższy zapewnia rozsądne wyniki.

```
niech h1 = 1
generuj hs+1 = 3hs + 1, dla s = 1, 2, 3, ...
zakończ, gdy ht+2   n; użyj h1 do ht jako odstępów w sortowaniu
```

Innymi słowy, generujemy kolejne elementy sekwencji, aż do momentu, gdy wartość wygenerowanego elementu przekroczy `n`. Następnie odrzucamy dwa ostatnie elementy sekwencji, a pozostałych używamy jako odstępów w sortowaniu.

Jeśli np. `n` = 100, generujemy następującą sekwencję: $h_1 = 1$, $h_2 = 4$, $h_3 = 13$, $h_4 = 40$, $h_5 = 121$. Ponieważ $h_5 > 100$, zatem do posortowania tablicy zawierającej 100 elementów używamy odstępów h_1, h_2 oraz h_3.

Złożoność algorytmu Shella waha się pomiędzy złożonością prostych metod sortowania (takich jak sortowanie przez wstawiania oraz przez wybieranie) wynoszącą $O(n^2)$ a złożonością $O(n\log_2 n)$ (charakterystyczną dla algorytmów sortowania przez kopcowanie, sortowania szybkiego oraz sortowania przez scalanie). Jej złożoność wynosi w przybliżeniu $O(n^{1,3})$ dla *n* należącego do praktycznego zakresu wielkości i zmierza do $O(n\log_2 n)$ przy *n* dążącym do nieskończoności.

W ramach ćwiczenia można napisać program sortujący listę przy użyciu sortowania Shella i zliczający ilość wykonanych operacji porównania i przypisania.

Kup ksi k Pole ksi k

Ćwiczenia

1. Napisz program umożliwiający porównanie wydajności metod sortowania przedstawionych w tym rozdziale pod względem „liczby porównań” oraz „liczby operacji przypisania”. W przypadku algorytmu sortowania szybkiego porównaj wydajność uzyskiwaną, gdy elementem rozdzielającym jest wybierany pierwszy element zakresu oraz element losowy.

   Sprawdź program, sortując: a) 10, 100, 1000, 10 000 oraz 1 000 000 elementów w losowej kolejności oraz b) 10, 100, 1000, 10 000 oraz 1 000 000 elementów już posortowanych.

2. Do funkcji `makeHeap` przekazywana jest tablica `A`. Jeśli `A[0]` zawiera `n`, to `A[1]` do `A[n]` zawiera liczby w losowej kolejności. Napisz funkcję `makeHeap` taką, że `A[1]` do `A[n]` zawiera kopiec maksymalny (z *największym* elementem w korzeniu). Funkcja musi przetwarzać elementy kopca w kolejności `A[2]`, `A[3]`, ..., `A[n]`.

3. Kopiec jest zapisany w jednowymiarowej tablicy liczb całkowitych `num[1..n]`, przy czym w pierwszej komórce zapisana jest *największa* wartość. Podaj wydajny algorytm, który pozwoli usunąć korzeń i przeorganizować elementy kopca w taki sposób, że będzie zajmował zakres tablicy od `num[1]` do `num[n-1]`.

4. Kopiec jest zapisany w jednowymiarowej tablicy liczb całkowitych `A[0...max]`, przy czym *największa* wartość znajduje się w komórce o numerze 1. Komórka `A[0]` określa bieżącą liczbę elementów kopca. Napisz funkcję, która doda do kopca nową wartość `v`. Funkcja powinna działać nawet wtedy, jeśli kopiec jest początkowo pusty, a w przypadku gdy nie będzie już w nim miejsca na dodanie nowego elementu, ma wyświetlać stosowny komunikat.

5. Napisz kod, który wczyta zbiór dodatnich liczb całkowitych (zakończony cyfrą `0`) i na ich podstawie utworzy w tablicy `H` kopiec, w którego korzeniu będzie umieszczona *najmniejsza* liczba. Po odczytaniu każda liczba jest dodawana do tablicy w taki sposób, że zostają zachowane właściwości kopca. W dowolnym momencie, jeśli wczytanych zostało *n* liczb, to `H[1..n]` musi zawierać kopiec. Można założyć, że tablica `H` jest na tyle duża, iż pomieści wszystkie liczby.

   Zakładamy, że zostały podane następujące liczby: `51 26 32 45 38 89 29 58 34 23 0`.

   Pokaż zawartość `H` po wczytaniu i przetworzeniu każdej z nich.

6. Do funkcji jest przekazywana tablica liczb całkowitych `A` oraz dwa indeksy `m` i `n`. Funkcja ma przeorganizować elementy od `A[m]` do `A[n]` i zwrócić indeks `d`, taki że wszystkie elementy położone na lewo od `d` będą mniejsze lub równe `A[d]`, a wszystkie elementy na prawo od `d` będą większe od `A[d]`.

7. Napisz funkcję, do której przekazywana jest tablica liczb całkowitych `num` oraz liczba całkowita `n` i która sortuje elementy tablicy od `num[1]` do `num[n]` przy użyciu sortowania Shella. Funkcja ma zwracać liczbę porównań wykonanych podczas sortowania. Do wyboru odstępów sortowania można wybrać dowolną rozsądną metodę.

8. W jednej tablicy liczb całkowitych `A[1..n]` w zakresie `A[1..k]` jest umieszczony kopiec minimalny, natomiast zakres `A[k+1..n]` zawiera dowolne wartości. Napisz wydajną funkcję, która scali całą zawartość tablicy w taki sposób, że będzie zawierać jeden kopiec minimalny. *Nie należy* przy tym używać żadnej dodatkowej tablicy.

9. W tablicy (np. `A`) jest zapisany kopiec maksymalny liczb całkowitych. Element `A[0]` zawiera rozmiar kopca (`n`), natomiast wartości kopca są umieszczone w elementach do `A[1]` do `A[n]`, przy czym w elemencie `A[1]` jest umieszczona *największa* wartość.

   i) Napisz funkcję `deleteMax`, która będzie usuwać z przekazanej tablicy `A` największy element, a następnie przeorganizuje tablicę w taki sposób, że jej zawartość pozostanie kopcem.

   ii) Przy założeniu, że dysponujesz dwoma tablicami `A` i `B`, takimi jak opisana powyżej, napisz kod, który scali ich zawartość i zapisze ją w trzeciej tablicy `C` w taki sposób, że będzie posortowana rosnąco. Metoda powinna porównywać po jednym elemencie z tablic `A` i `B`. Można przy tym założyć, że dostępna jest funkcja `deleteMax`.

10. Napisz funkcję rekurencyjną umożliwiającą znalezienie *k*-tej najmniejszej wartości w tablicy zawierającej *n* liczb, bez sortowania jej zawartości.

Kup ksi k Pole ksi k

11. Napisz funkcję sortowania przez wstawianie, korzystającą z wyszukiwania binarnego w celu określenia miejsca, w którym element `A[j]` ma zostać wstawiony do posortowanej podlisty `A[1..j-1]`.
12. Mówimy, że algorytm sortowania jest *stabilny*, jeśli proces sortowania zachowuje względną kolejność kluczy. Które z opisanych metod sortowania są stabilne?
13. Dysponujemy listą *n* liczb. Napisz wydajny algorytm umożliwiający znalezienie: a) wartości najmniejszej, b) wartości największej, c) średniej, d) mediany oraz e) dominanty (wartości występującej najczęściej).

    Napisz wydajny algorytm znajdujący te wartości.
14. Wiemy, że każda liczba znajdująca się na liście *n* *unikalnych* liczb jest z zakresu od 100 do 999. Wymyśl wydajny algorytm sortowania takiej listy.

    Zmodyfikuj podany algorytm w taki sposób, by sortował także listę zawierającą powtarzające się liczby.
15. Zmodyfikuj algorytmy sortowania przez scalanie (przedstawione w rozdziale 5.) oraz sortowania szybkiego w taki sposób, że podlisty zawierające mniej niż określoną, predefiniowaną liczbę elementów będą sortowane przy użyciu sortowania przez wstawianie.
16. Dysponujemy listą `n` liczb oraz dodatkową liczbą `x`. Należy znaleźć najmniejszą liczbę zapisaną na liście mniejszą lub równą `x`. Odnalezioną liczbę należy usunąć z listy, a na jej miejsce wstawić nową liczbę `y`, zachowując przy tym strukturę listy. Wymyśl rozwiązania tego problemu, wykorzystując: a) nieposortowaną tablicę liczb, b) posortowaną tablicę liczb, c) posortowaną listę powiązaną, d) binarne drzewo poszukiwań oraz e) kopiec.

    Która z tych metod jest najbardziej wydajna?
17. Dysponujemy (długą) listą słów. Napisz program pozwalający określić, które z tych słów są anagramami. Wyniki prezentowane przez program powinny zawierać każdą podlistę odnalezionych anagramów (dwa lub więcej słów), przy czym poszczególne podlisty mają być od siebie oddzielone jednym pustym wierszem. Słowa są anagramami, jeśli składają się z tych samych liter, np. kaszel i szekla, kaprys i pryska.
18. Każda wartość tablicy `A[1..n]` jest liczbą 1, 2 lub 3. Podaj *minimalną* liczbę operacji *zamiany*, które trzeba wykonać w celu posortowania takiej tablicy. Przykładowo tablicę

2	2	1	3	3	3	2	3	1
1	2	3	4	5	6	7	8	9

    można posortować, wykonując cztery zamiany, w następującej kolejności: (1, 3), (4, 7), (2, 9) oraz (5, 9). Innym rozwiązaniem są zamiany: (1, 3), (2, 9), (4, 7) i (5, 9). Tablicy nie można posortować, wykonując mniej niż cztery operacje zamiany.

Kup ksi k

Pole ksi k

Kup ksi k

Pole ksi k

Skorowidz

A

abstrakcyjne typy danych, 123
adres
    obiektu, 61–63
    pierwszego węzła, 94
adresowanie otwarte, 306
akcesor, 52, 53
aktualizowanie
    pliku, 221
    wskaźnika, 89
algorytm
    Euklidesa, 156
    haszowania, 307
    scalania, 39
    sortowanie przez wstawianie, 20, 25
    sortowanie przez wybieranie, 19
atrybut, 44
automatyczne odzyskiwanie pamięci, 95

B

binarne drzewo poszukiwań, 237–240, 247, 255
blok
    catch, 203
    try, 203
błąd przepełniania buforu rekurencji, 280
brat, sibling, 226
BTS, binary search trees, 237
budowanie
    drzewa binarnego, 233, 240, 244, 247
    kopca, 269

C

ciało
    klasy, 45
    konstruktora, 49
ciąg Fibonacciego, 156

D

definicje rekurencyjne, 153
definiowanie
    klas, 44
    list powiązanych, 83
długość ścieżki wewnętrznej, 260
dodawanie
    dwóch liczb, 181
    elementów do kolejki, 94
    elementów do listy, 88, 93
    nowego elementu, 26
dostęp
    do obiektu, 61
    do zmiennej, 46
droga w labiryncie, 170
drzewo, 225, 226
    binarne, 225, 231, 244
        kompletne, 257
        pełne, 259
        prawie kompletne, 257, 259
        ze wskaźnikami rodzica, 244
    uporządkowane, 226
    zdegenerowane, 239
    zorientowane, 226
dynamiczne przydzielanie pamięci, 88
dziecko, child, 226

E

element rozdzielający, pivot, 274

F

funkcja, *Patrz* metoda
funkcja mieszająca, hash function, 293, 296, 305
funkcje rekurencyjne, 154

G

generowanie liczb losowych, 179
głębokość wierzchołka, 226
gra
  Nim, 182
  Węże i drabiny, 178
  zgadywanka, 180
grupowanie, clustering, 298
  danych, 71
  pierwotne, 298, 307
  wtórne, 298

H

haszowanie, 291, 307
hermetyzacja danych, 51

I

implementacja
  kolejki
    listą powiązaną, 147
    tablicą, 143
  metody łańcuchowej, 303
  scalania list, 38
  stosu
    listą powiązaną, 127
    tablicą, 124
indeks, 214
inicjalizowanie
  pola, 47, 48
  zmiennych, 46
instancja klasy, 44
instrukcja return, 63

J

język obiektowy, 44

K

kapsel, 186
klasa, 44
  Arithmetic, 182
  BinarySearchString, 33
  BinarySearchTest, 32
  BinarySearchTreeTest, 247
  BinaryTree, 232, 234, 248, 252
  BinaryTreeTest, 235
  Book, 45
  BottleCaps, 187
  BuildList1, 90
  BuildList2, 93
  BuildList3, 97
  CompareFiles, 201
  CreateBinaryFile, 205–208
  CreateIndex, 215
  CreateRandomAccess, 211
  DataInputStream, 205, 207
  DataOutputStream, 205
  DecimalToBinary, 133
  DistinctNumbers, 295
  EvalExpression, 140
  Exception, 203, 204
  GuessTheNumber, 180
  HashChain, 301
  HeapSortTest, 268
  Index, 214, 217
  InfixToPostfix, 137
  InsertSort1Test, 23, 25
  InsertSort2Test, 26
  LevelOrderTest, 252
  LinkedList, 99–106
  Maze, 172
  MergeLists, 115
  MergeSortTest, 165
  MergeTest, 39
  Nim, 184
  Node, 85, 99, 106, 130, 138
  NodeData, 100, 109, 131, 148, 232, 240, 248
  Organisms, 168
  Palindrome, 110
  ParallelSort, 29
  Part, 57, 65, 208, 216
  PartTest, 58
  Person, 67, 75, 78
  Pi, 195
  QNode, 250, 253
  Queue, 143, 146, 251
  QueueData, 250, 253
  QueueTest, 146, 149

Kup ksi k   Pole ksi k

Quicksort3Test, 280
QuicksortTest, 276
RandomAccessFile, 209
RandomTest, 179
ReadBinaryFile, 206
ReadRandomAccess, 212
Root5, 193
Scanner, 45, 241
SearchTest, 68
SelectSortTest, 18
ShellSortTest, 286
SiftUpTest, 271
SimulateQueue, 191
SortStrings, 28
Stack, 127–132, 138
StackTest, 126, 129, 132
String, 27, 45
Sum, 44
TreeNode, 231, 248
UseIndex, 218
VoteCount, 79
Voting, 77
WordFrequency, 34, 72
WordFrequencyBST, 242
WordFrequencyHash, 308
WordInfo, 71, 73
klasy użytkownika, 51
klucz, 292
kolejka, 94, 142
kolejka priorytetowa, 273
kolizja, 291, 293, 297
konstrukcja try…catch, 202
konstruktor, 49
bezargumentowy, 48, 50
domyślny, 48
poprawiony, 53
konwersja
drzewa binarnego, 264
liczb, 133, 156
wyrażenia, 134
kopiec, 264, 273
maksymalny, 264
minimalny, 264
korzeń, root, 225

L

las, 226
liczba
porównań, 25
wierzchołków drzewa, 254
π, 194
liczby
bliźniacze, 306
losowe, 177
pseudolosowe, 178
licznik, 71, 86
licznik występowania słów, 307
lista, 37
dwukierunkowa, 119
cykliczna, 116, 144
jednokierunkowa, 91
jednokrotnie powiązana, 91
liniowa, 142
powiązana, 83, 111
liść, leaf, 226

M

metaznaki, 242
metoda
addHead, 100
addInPlace, 102
addTai, 106
addToTable, 309
binarySearch, 31–33, 37
buildTree, 233–236, 245
compareTo, 27, 33, 102, 240
compareToIgnoreCase, 27
copyList, 106
countLeaves, 254
countNodes, 254
createMasterIndex, 215
decToBin, 157
deleteNode, 256
dequeue, 145, 148, 150
enqueue, 145, 150
equals, 27, 107
equalsIgnoreCase, 27, 67
fact, 155
findOrg, 168
findOrInsert, 243
findPath, 172
getPhrase, 109
getSmallest, 17
getToken, 136
getWord, 37
hanoi, 161
heapSort, 267
height, 254
hsort, 286
insertInPlace, 26
insertionSort, 23–28
kthSmall, 283
levelOrderTraversal, 253

Kup ksi k

Pole ksi k

metoda
  łańcuchowa, 300
  main, 37
  merge, 39, 163
  mergeSort, 164
  parallelSort, 29
  partition, 283
  partition2, 279
  peek, 137
  pop, 129, 131
  power, 162
  precedence, 136
  printFileInOrder, 220
  printList, 91
  processVotes, 77
  push, 128, 131
  quicksort, 275
  quicksort2, 279
  quicksort3, 282
  random, 179
  readPartFromFile, 219
  reverseList, 107
  search, 301, 308
  seek, 210
  selectionSort, 18
  siftDown, 266
  siftUp, 269, 272
  smallest, 192
  swap, 17
  toString, 56, 101
  updateRecord, 222
  visit, 241
  writePartToFile, 216, 223
metody
  instancyjne, 45, 55
  klasowe, 45
  klasy BinaryTree, 254
  klasy DataInputStream, 207
  niestatyczne, 45
  statyczne, 45, 56
mieszanie, *Patrz* haszowanie
modyfikator dostępu, 45
moment drzewa, 226
mutator, 52, 54

N

nagłówek klasy, 45
następnik, successor, 245
nazwy plików, 59
  wewnętrzne, 200
  zewnętrzne, 200
notacja
  przyrostkowa, 134, 139
  wrostkowa, 134

O

obiekt, 44, 60
obsługa klientów, 190
odczyt z pliku, 199
odkładanie na stos, 94
odnajdywanie drogi, 170
odwołania do zmiennych, 46
odwracanie listy powiązanej, 159
odzyskiwanie pamięci, 95
operacje
  na listach, 85
  wejścia-wyjścia, 199, 205
ostatni węzeł listy, 87

P

palindrom, 107
pamięć statyczna, 88
pętla
  do…while, 185
  for, 17
  while, 23, 40
pierwszeństwo operatorów, 134
plik
  btree.in, 236
  LinkedList.java, 105
  parts.txt, 217
pliki
  binarne, 200, 205
  binarne z rekordami, 207
  indeksowane, 213
  o dostępie swobodnym, 209, 221
  tekstowe, 200
  źródłowe, 104
poddrzewo, 225
podwójne haszowanie, 306
podział tablicy, 277
pole, 44
  head, 99, 148
  next, 84
  tail, 148
porównywanie
  list, 108
  łańcuchów znaków, 27
  plików, 201
  zmiennych obiektowych, 62

Kup ksi k Pole ksi k

problem
ośmiu królowych, 175
wyszukiwania i wstawiania, 291, 292
próbkowanie
kwadratowe, 299
liniowe, linear probing, 297
liniowe z podwójnym haszowaniem, 306
przechodzenie
drzewa, 228, 231, 235, 244
drzewa poziomami, 249
metodą
in-order, 229
level-order, 251
post-order., 230
pre-order, 228, 259
poprzeczne, 229
wsteczne, 229
wzdłużne, 228
przechowywanie list, 111, 112
przeciążanie konstruktorów, 50
przekazywanie obiektu jako argumentu, 64
przepełnienie stosu, 125
przesiewanie w górę, 270
przeszukiwanie
listy powiązanej, 86
tablicy, 32, 34, 67
przetwarzanie list powiązanych, 83
przypisywanie zmiennych obiektowych, 60
publiczna metoda akcesora, 52
punkt podziału, division point, 274

R

referencja, 47
rekurencja, 153
rekurencyjna definicja funkcji, 154
reprezentacja drzewa binarnego, 231
rodzic, parent, 226
rozkład
nierównomierny, 186
równomierny, 177
rozszerzanie klasy, 106
rozwiązywanie kolizji, 293
metoda łańcuchowa, 300
podwójne haszowanie, 306
próbkowanie kwadratowe, 299
próbkowanie liniowe, 298

S

scalanie list, 37, 113, 163
silnia, 153
słowo kluczowe
class, 45
new, 45
null, 63
private, 46
protected, 46
public, 46
static, 45, 46
this, 102
sortowanie, 15
listy, 95, 112
przez
kopcowanie, heapsort, 263, 272
scalanie, 163
wstawianie, 19, 70
wybieranie, 15, 19, 273
Shella, shellsort, 284
szybkie, quicksort, 274, 277, 280
tablicy, 15, 19
łańcuchów znaków, 27
obiektów, 70
równoległej, 29
z użyciem malejących odstępów, 284
stała
MaxWords, 37
StringFixedLength, 211
stan obiektu, 44
stopień, degree, 225
stos, 124
operatorów, 135
pusty, 128
wykonawczy, 155
stosowanie obiektów, 60
strumień wyjściowy, 206, 241
symulowanie
kolejki, 190
realnych problemów, 189
zbierania kapsli, 188
szacowanie
pierwiastka kwadratowego, 193
wartości π, 194

T

tablica, 15, 111
candidate, 76
łańcuchów znaków, 27, 32, 70
mieszająca, 295, 296
nadmiarowa, overflow table, 303
typu String, 27
tablice
obiektów, 64, 69, 70
równoległe, 29

Kup ksi k
Pole ksi k

tablicowa reprezentacja drzewa, 257
testowanie klasy Part, 58
tryb rw, 221
tworzenie
klasy listy powiązanej, 99
listy posortowanej, 95
listy powiązanej, 88, 93
obiektów, 44
węzła, 94
typ
LinkedList, 114
Stack, 130

U

umieszczanie na stosie, 124
usuwanie
elementu z listy, 94
elementu z tablicy, 296
kluczy, 304
wierzchołków, 255

W

waga drzewa, 226
wartość
domyślna, 48
END, 211
key, 221, 295
wyrażenia, 139
węzły listy, 83
wierzchołek końcowy, 226
wieże Hanoi, 160
wskaźnik, 47
do węzła, 84, 95
na korzeń, 233
na rodzica, 247
null, 63, 84
współczynnik wypełnienia tablicy, 299
wstawianie
elementu, 26, 91
węzła do listy, 91
wyjątek, 203, 204
wyliczanie
potęgi liczby, 162
wartości wyrażeń, 139
wyrażenie
w zapisie przyrostkowym, 142
w zapisie wrostkowym, 142
wysokość drzewa, 226
wyszukiwanie
binarne, 30
i wstawianie, 291, 292
łańcucha, 68
wyświetlanie
listy powiązanej, 159
zawartości pól, 55

Z

zapis
do pliku, 36, 199, 206
listy powiązanej, 111
przyrostkowy, 134, 141
wrostkowy, 134, 140
zdejmowanie ze stosu, 94, 124
zliczanie, 166
węzłów, 85
wystąpień wyrazów, 34, 240
złożoność algorytmu, 25, 287
zmienna
curr, 85, 95
obiektowa, 47
prev, 95
top, 84, 93, 125–128
zmienne
instancyjne, 44, 51
klasowe, 45, 51
niestatyczne, 45
obiektowe, 60, 62
statyczne, 45
znajdowanie najmniejszej liczby, 283
zwracanie wartości, 74

Kup ksi k
Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA

POZNAJ ZAAWANSOWANE MOŻLIWOŚCI JĘZYKA JAVA!

Java
Zaawansowane zastosowania

Czy wiesz, jaki język programowania wybierany jest jako podstawa najbardziej skomplikowanych i zaawansowanych projektów IT? Tak, to Java! Doskonale sprawdza się wszędzie tam, gdzie wymagane są najwyższa wydajność, pełne bezpieczeństwo oraz realizacja złożonych reguł biznesowych. Jeżeli chcesz zapoznać się z nietypowym i sprytnym wykorzystaniem tego języka, to trafiłeś na doskonałą książkę.

W trakcie lektury będziesz mieć niepowtarzalną okazję, by przygotować zaawansowane algorytmy oraz zaimplementować je z użyciem języka Java. Ponadto dogłębnie poznasz listy, stosy i kolejki oraz dowiesz się, jak efektywnie na nich operować. W kolejnych rozdziałach zaznajomisz się z technikami sortowania danych oraz generowania liczb losowych. Co jeszcze? Operacje na plikach, drzewa binarne oraz haszowanie to tylko niektóre z poruszanych tu tematów. Książka jest doskonałą lekturą dla wszystkich programistów języka Java, chcących wycisnąć z niego jeszcze więcej!

Dzięki tej książce:

- poznasz i zaimplementujesz zaawansowane algorytmy,
- wygenerujesz liczby losowe,
- poznasz zaawansowane metody sortowania danych,
- zaznajomisz się z tematem rekurencji,
- poznasz niuanse języka Java.

Apress

sięgnij po WIĘCEJ

KOD KORZYŚCI

Nr katalogowy: 23857

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900
0 601 339900

helion.pl księgarnia internetowa

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

cena: 59,00 zł

ISBN 978-83-246-9426-6

9 788324 694266

Informatyka w najlepszym wydaniu