James Cutajar

Struktury danych i algorytmy w języku Java

Przewodnik dla początkujących

Helion

Packt›

Tytuł oryginału: Beginning Java Data Structures and Algorithms

Tłumaczenie: Krzysztof Bąbol

ISBN: 978-83-283-5329-9

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

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

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

Printed in Poland.

Spis treści

O autorze 7

Wstęp 9

Rozdział 1. Algorytmy i ich złożoność 13

Tworzymy nasz pierwszy algorytm 14
Algorytm konwersji liczb dwójkowych na dziesiętne 14
Mierzenie złożoności algorytmów za pomocą notacji dużego O 16
Przykład na złożoność 16
Zrozumienie złożoności 19
Notacja złożoności 22
Identyfikacja algorytmów o różnej złożoności 26
Złożoność liniowa 27
Złożoność kwadratowa 28
Złożoność logarytmiczna 29
Złożoność wykładnicza 30
Złożoność stała 32
Podsumowanie 34

Rozdział 2. Algorytmy sortowania i podstawowe struktury danych 35

Wprowadzenie do sortowania bąbelkowego 35
Zrozumienie sortowania bąbelkowego 36
Udoskonalanie sortowania bąbelkowego 37
Zrozumienie sortowania szybkiego 40
Zrozumienie rekurencji 40
Podział w wyszukiwaniu szybkim 42
Jak to wszystko poskładać razem 44
Korzystanie z sortowania przez scalanie 46
Dzielenie problemu 47
Scalanie problemu 48

Rozpoczęcie pracy z podstawowymi strukturami danych **50**
Wprowadzenie do struktur danych 51
Struktura list powiązanych 51
Operacje na listach powiązanych 53
Kolejki 57
Stosy 58
Modelowanie stosów i kolejek przy użyciu tablic 60
Podsumowanie **64**

Rozdział 3. Tablice z haszowaniem i binarne drzewa poszukiwań **65**

Wprowadzenie do tablic z haszowaniem **65**
Zrozumienie tablic z haszowaniem 66
Rozwiązywanie kolizji przez łańcuchowanie 68
Rozwiązywanie kolizji przez adresowanie otwarte 71
Haszowanie uniwersalne 76
Rozpoczęcie pracy z binarnymi drzewami poszukiwań **78**
Struktura drzewa binarnego 78
Operacje na binarnych drzewach poszukiwań 80
Przechodzenie przez binarne drzewo poszukiwań 84
Zrównoważone binarne drzewa poszukiwań 86
Podsumowanie **91**

Rozdział 4. Paradygmaty projektowania algorytmów **93**

Wprowadzenie do algorytmów zachłannych **94**
Problem wyboru zajęć 94
Rozwiązanie problemu wyboru zajęć 96
Składniki algorytmu zachłannego 96
Kodowanie Huffmana 99
Ćwiczenie: Implementacja algorytmu zachłannego do obliczania ułamków egipskich 102
Wprowadzenie do algorytmów typu „dziel i zwyciężaj” **103**
Podejście „dziel i zwyciężaj” 103
Metoda rekurencji uniwersalnej 104
Problem najbliższej pary punktów 106
Ćwiczenie: Rozwiązywanie problemu podtablicy o największej sumie 109
Zrozumienie programowania dynamicznego **110**
Elementy problematyki programowania dynamicznego 110
Dyskretny problem plecakowy 111
Najdłuższy wspólny podciąg 114
Ćwiczenie: Problem wydawania reszty 116
Podsumowanie **117**

Rozdział 5. Algorytmy wyszukiwania wzorca w tekście **119**

Algorytm wyszukiwania naiwnego **119**
Implementacja wyszukiwania naiwnego 120
Usprawnienie algorytmu wyszukiwania naiwnego 121

Pierwsze kroki z algorytmem wyszukiwania wzorca Boyera-Moore'a **122**
Zasada niezgodności 123
Zasada dobrego sufiksu 125
Zastosowanie algorytmu Boyera-Moore'a 129
Prezentacja innych algorytmów wyszukiwania wzorca w tekście **130**
Algorytm Rabina-Karpa 130
Algorytm Knutha-Morrisa-Pratta 132
Algorytm Aho-Corasick 132
Podsumowanie **133**

Rozdział 6. Grafy, liczby pierwsze i klasy złożoności **135**

Reprezentacja grafów **136**
Listy sąsiedztwa 137
Macierz sąsiedztwa 139
Przechodzenie przez graf **142**
Przeszukiwanie wszerz 142
Przeszukiwanie w głąb 144
Wykrywanie cykli 147
Obliczanie najkrótszych ścieżek **149**
Najkrótsza ścieżka z pojedynczego źródła: algorytm Dijkstry 150
Najkrótsze ścieżki dla wszystkich par wierzchołków: algorytm Floyda-Warshalla 154
Liczby pierwsze w algorytmach **158**
Sito Eratostenesa 158
Rozkład na czynniki pierwsze 158
Inne koncepcje związane z grafami **159**
Minimalne drzewa rozpinające 159
Algorytm A* 160
Problem maksymalnego przepływu 161
Zrozumienie klas złożoności problemów **161**
Podsumowanie **162**

Skorowidz **163**

ROZDZIAŁ 2

Algorytmy sortowania i podstawowe struktury danych

W poprzednim rozdziale pokazałem, jak ulepszyć rozwiązanie problemu znajdowania części wspólnej, korzystając z algorytmu sortowania. To częsty przypadek. Jeżeli bowiem dane są uporządkowane, można opracować efektywniejszy algorytm. Ten rozdział rozpoczniemy od zbadania trzech technik sortowania: bąbelkowego, szybkiego i przez scalanie. Później poznasz różne sposoby organizowania danych w podstawowe struktury.

Z tego rozdziału dowiesz się, jak:

- opisać sposób działania sortowania bąbelkowego;
- zaimplementować lepsze rozwiązanie — sortowanie szybkie;
- scharakteryzować sortowanie przez scalanie;
- stworzyć strukturę danych — listę wiązaną;
- zaimplementować kolejki;
- opisać strukturę danych stosu.

Wprowadzenie do sortowania bąbelkowego

Sortowanie bąbelkowe jest najprostszym algorytmem sortowania. Wiąże się z wielokrotnym przejściem przez tablicę wejściową i zamianą miejscami sąsiadujących nieuporządkowanych elementów. Technika ta nazywa się sortowaniem bąbelkowym, ponieważ posortowane fragmenty jak bąbelki przemieszczają się z końca listy.

Zrozumienie sortowania bąbelkowego

Każdy algorytm sortowania przyjmuje listę elementów i zwraca je uporządkowane. Główną różnicą między algorytmami jest sposób, w jaki dokonują sortowania. Sortowanie bąbelkowe działa na zasadzie zamiany miejscami sąsiadujących elementów. Tak posortowane elementy są spychane na koniec listy.

Na listingu 2.1 został pokazany pseudokod sortowania bąbelkowego. Algorytm wykonuje trzy proste zadania: wielokrotnie przechodzi przez listę do sortowania, porównuje sąsiadujące elementy i zamienia je miejscami, jeśli pierwszy element jest większy od drugiego.

Listing 2.1. Pseudokod sortowania bąbelkowego

```
bubbleSort(array)
  n = length(array)
  for (k = 1 until n)
    for (j = 0 until -1)
      if(array[j] > array[j + 1])
        swap(array, j, j + 1)
```

Ile przejść należy wykonać, by tablica była posortowana? Jak się okazuje, potrzeba do tego n–1 iteracji, gdzie n jest wielkością tablicy. W następnym punkcie pokażę, dlaczego konieczne jest tyle przejść, ale sortowanie bąbelkowe ma złożoność czasową $O(n^2)$ właśnie dlatego, że przetwarzamy n elementów n–1 razy.

> Funkcja swap z listingu 2.1 przy użyciu zmiennej tymczasowej zamienia wartości w miejscach tablicy o indeksach j oraz j+1.

Implementacja sortowania bąbelkowego

Aby zaimplementować sortowanie bąbelkowe w Javie, wykonaj następujące kroki:

1. Przełóż pseudokod pokazany na listingu 2.1 na język Java. Utwórz klasę i metodę przyjmującą do sortowania tablicę w następujący sposób:

   ```
   public void sort(int[] numbers)
   ```

2. Trudność w tym algorytmie może sprawić zamiana elementów miejscami. W tym celu jeden z zamienianych elementów przypisuje się do zmiennej tymczasowej, tak jak na listingu 2.2.

Listing 2.2. Metoda sortowania bąbelkowego. Nazwa klasy źródłowej: BubbleSort

```
public void sort(int[] numbers) {
  for (int i = 1; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length - 1; j++) {
      if (numbers[j] > numbers[j + 1]) {
        int temp = numbers[j];
```

```
            numbers[j] = numbers[j + 1];
            numbers[j + 1] = temp;
          }
        }
      }
    }
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

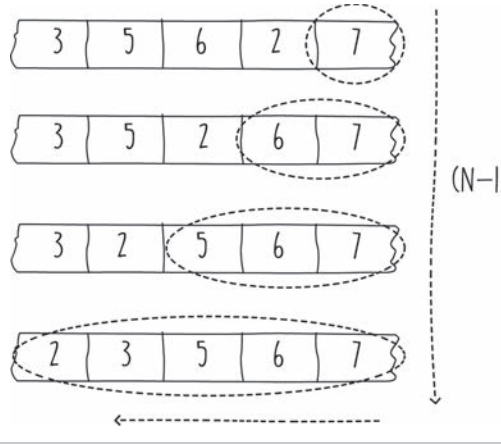
Chociaż sortowanie bąbelkowe bardzo łatwo zaimplementować, jest też jedną z najwolniejszych metod sortowania. W następnej sekcji przyjrzymy się, jak poprawić nieco wydajność tego algorytmu.

Udoskonalanie sortowania bąbelkowego

Aby poprawić wydajność sortowania bąbelkowego, można zastosować dwie główne techniki. Trzeba zdawać sobie sprawę, że chociaż w przeciętnym przypadku obie te strategie poprawiają ogólną wydajność sortowania bąbelkowego, to w najgorszym przypadku algorytm nadal ma tę samą słabą złożoność czasową $O(n^2)$.

Pierwszym drobnym ulepszeniem, jakie można wprowadzić do pierwotnego sortowania bąbelkowego, jest wykorzystanie faktu, że uporządkowany „bąbelek" tworzy się na końcu listy. Podczas każdego przejścia do „bąbelka" dołączany jest kolejny element. Dlatego właśnie wymagane jest $(n–1)$ przejść.

Przedstawia to również rysunek 2.1, gdzie elementy w kropkowanym kole są już posortowane i we właściwym miejscu.



Rysunek 2.1. Tworzenie „bąbelków" na końcu listy

Można wykorzystać ten fakt, by nie sortować elementów wewnątrz „bąbelka". W tym celu należy nieznacznie zmodyfikować kod w Javie w sposób pokazany na listingu 2.3. W wewnętrznej pętli, zamiast przetwarzać listę do końca, możemy zatrzymać się tuż przed posortowanym „bąbelkiem", na pozycji `numbers.length - i`. Dla zwięzłości w listingu 2.3 zamiast kodu dokonującego zamiany elementów miejscami wprowadziłem wywołanie metody w inny sposób.

Listing 2.3. Ulepszone sortowanie bąbelkowe 1. Nazwa klasy źródłowej: BubbleSort

```
public void sortImprovement1(int[] numbers) {
  for (int i = 1; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length - i; j++) {
      if (numbers[j] > numbers[j + 1]) {
        swap(numbers, j, j + 1);
      }
    }
  }
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Jeśli do algorytmu sortowania bąbelkowego przekażemy posortowaną listę, to mimo braku zmian nadal będziemy wykonywać wiele przejść. Możemy więc poprawić algorytm, tak by wychodził z pętli zewnętrznej, jeżeli lista wewnątrz tablicy jest w pełni posortowana. W tym celu należy sprawdzić, czy podczas ostatniego przejścia dokonano jakichkolwiek zamian. Jeśli więc przekażemy do metody już posortowaną listę, wystarczy przejść jeden raz przez tablicę i pozostawić ją bez zmian. Oznacza to, że przypadek optymistyczny ma teraz złożoność O (n), chociaż złożoność pesymistyczna pozostała taka sama.

Implementacja poprawionego sortowania bąbelkowego

Ulepszymy algorytm sortowania bąbelkowego, zmniejszając liczbę przebiegów.

W tym celu trzeba wykonać następujące kroki:

1. Zmień metodę sortowania bąbelkowego, aby przestała sortować, jeśli po przejściu przez wewnętrzną pętlę tablica nie uległa zmianom.
2. Rozwiązanie to najłatwiej opracować, zamieniając zewnętrzną pętlę `for` na pętlę `while` oraz wprowadzając flagę wskazującą, czy podczas przechodzenia przez tablicę zostały zamienione jakieś elementy. Pokazuje to listing 2.4.

Listing 2.4. Ulepszone sortowanie bąbelkowe 2. Nazwa klasy źródłowej: BubbleSort

```
public void sortImprovement2(int[] numbers) {
  int i = 0;
  boolean swapOccured = true;
  while (swapOccured) {
```

```
            swapOccured = false;
            i++;
            for (int j = 0; j < numbers.length - i; j++) {
                if (numbers[j] > numbers[j + 1]) {
                    swap(numbers, j, j + 1);
                    swapOccured = true;
                }
            }
        }
    }
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

W tym punkcie pokazałem kilka prostych sztuczek poprawiających algorytm sortowania bąbelkowego. W kolejnych podrozdziałach przyjrzymy się innym technikom sortowania, które działają znacznie szybciej niż sortowanie bąbelkowe.

Ćwiczenie: Implementacja sortowania przez wybieranie w języku Java

Scenariusz

Sortowanie przez wybieranie najłatwiej zrozumieć na podstawie dwóch list, A i B. Na początku lista A zawiera wszystkie nieposortowane elementy, lista B jest zaś pusta. Pomysł polega na zapisywaniu posortowanych elementów w liście B. Działanie algorytmu polega na znajdowaniu najmniejszego elementu listy A i przesuwaniu go na koniec listy B. Robimy to, dopóki lista A nie jest pusta, a B nie jest pełna.

Zamiast dwóch oddzielnych list możemy po prostu użyć tej samej tablicy wejściowej, utrzymując wskaźnik dzielący tablicę na dwie części.

Można to wyjaśnić na przykładzie z życia. Wyobraź sobie sortowanie talii kart. Po potasowaniu talii przeglądasz karty jedna po drugiej, aż znajdziesz najniższą. Odkładasz ją na bok na nowy, drugi stos. Potem szukasz następnej najniższej karty i po znalezieniu kładziesz ją na spód drugiego stosu. Powtarzasz te czynności, dopóki pierwszy stos nie jest pusty.

Jednym ze sposobów dojścia do rozwiązania jest napisanie najpierw pseudokodu, korzystającego z dwóch tablic (opisanych wcześniej jako A i B). Następnie należy dostosować pseudokod tak, by zapisywał posortowaną listę (tablicę B) w tej samej tablicy, w której są dane wejściowe.

Cel

Implementacja sortowania przez wybieranie w języku Java.

Warunki wstępne

- Zaimplementuj metodę `sort`, znajdującą się w klasie dostępnej w kodzie dołączonym do tej książki pod adresem *src/main/java/com/packt/datastructuresandalg/lesson2/activity/selectionsort/SelectionSort.java*
- Metoda `sort` powinna przyjąć tablicę liczb całkowitych i ją posortować.

Jeśli skonfigurowałeś projekt, testy jednostkowe do tego ćwiczenia uruchomisz następującym poleceniem:

```
gradlew test --tests com.packt.datastructuresandalg.lesson2.activity.selectionsort*
```

Kroki do wykonania

1. Podziel tablicę wejściową na dwie za pomocą wskaźnika indeksu tablicy.
2. Metoda `sort` powinna przyjąć tablicę liczb całkowitych i ją posortować.
3. Przejdź przez nieposortowaną część tablicy, aby znaleźć w niej wartość minimalną.
4. Zamień znaleziony element miejscami, przenosząc go na koniec części posortowanej.

Zrozumienie sortowania szybkiego

Sortowanie szybkie stanowi duży postęp w porównaniu z sortowaniem bąbelkowym. Technika sortowania szybkiego została opracowana przez brytyjskiego informatyka Tony'ego Hoare'a. Algorytm wykonuje trzy główne kroki:

1. Wybiera element osiowy.
2. Dzieli listę tak, by elementy po lewej stronie elementu osiowego były mniejsze niż wartość tego elementu, a te po prawej większe.
3. Powtarza kroki 1. i 2. osobno dla lewej i prawej części.

Ponieważ sortowanie szybkie wymaga użycia rekurencji, ten podrozdział zaczniemy od zaprezentowania jej przykładu. Potem przyjrzymy się dokonywaniu podziału w algorytmie sortowania szybkiego, a w końcowej części zastosujemy techniki rekurencyjne.

Zrozumienie rekurencji

Rekurencja to naprawdę przydatne narzędzie dla twórców algorytmów. Pozwala pokonywać duże problemy, rozwiązując te same problemy w mniejszej skali. Funkcje rekurencyjne mają zwykle podobną strukturę i zawierają następujące składniki:

- *Co najmniej jeden warunek zatrzymania*: w pewnych warunkach powstrzymują one funkcję przed ponownym wywołaniem siebie samej.
- *Co najmniej jedno wywołanie rekurencyjne*: mają one miejsce wtedy, gdy funkcja (lub metoda) wywołuje samą siebie.

W następnym przykładzie weźmiemy na warsztat problem wyszukiwania binarnego z poprzedniego rozdziału i zmienimy algorytm tak, by działał rekurencyjnie. Przeanalizujmy problem wyszukiwania binarnego omówiony w rozdziale 1., „Algorytmy i i ich złożoność", pokazany na listingu 1.7. Implementacja ta jest iteracyjna, to znaczy działa w pętli, aż element zostanie znaleziony albo parametr `end` będzie równy lub większy od zmiennej `start`. Listing 2.5 zawiera pseudokod pokazujący, jak możemy zamienić tę metodę na funkcję rekurencyjną.

Listing 2.5. Rekurencyjny pseudokod wyszukiwania binarnego

```
binarySearch(x, array, start, end)
  if(start <= end)
    mid = (end - start) / 2 + start
    if (array[mid] == x) return true
    if (array[mid] > x) return binarySearch(x, array, start, mid - 1)
    return binarySearch(x, array, mid + 1, end)
  return false
```

W rekurencyjnym wyszukiwaniu binarnym tak naprawdę istnieją dwa warunki zatrzymania. Funkcja zatrzymuje łańcuch rekurencji, jeśli po drodze znajdzie wyszukiwany element albo jeśli wskaźnik początkowy do tablicy będzie większy od końcowego, co oznacza, że element nie został znaleziony. Warunek zatrzymania można łatwo znaleźć, sprawdzając wszystkie drogi wyjścia, które nie wymagają dalszych wywołań rekurencyjnych.

Implementacja rekurencyjnego wyszukiwania binarnego

Aby zaimplementować rekurencyjne wyszukiwanie binarne w języku Java, wykonaj następujące kroki:

1. Używając pseudokodu pokazanego na listingu 2.5, zaimplementuj funkcję wykonującą rekurencyjnie wyszukiwanie binarne.
2. Dodaj kolejną metodę, której sygnatura będzie zawierać tylko szukany element i posortowaną tablicę jako dane wejściowe. Ta metoda wywoła funkcję rekurencyjną z odpowiednimi wartościami w następujący sposób:

```
public boolean binarySearch(int x, int[] sortedNumbers)
```

Wynik

Ta dodatkowa metoda wywołuje wstępnie funkcję rekurencyjną, pokazaną na listingu 2.6.

Listing 2.6. Rekurencyjne wyszukiwanie binarne. Nazwa klasy źródłowej: BinarySearchRecursive

```
public boolean binarySearch(int x, int[] sortedNumbers, int start,
int end) {
  if (start <= end) {
    int mid = (end - start) / 2 + start;
    if (sortedNumbers[mid] == x) return true;
    if (sortedNumbers[mid] > x)
```

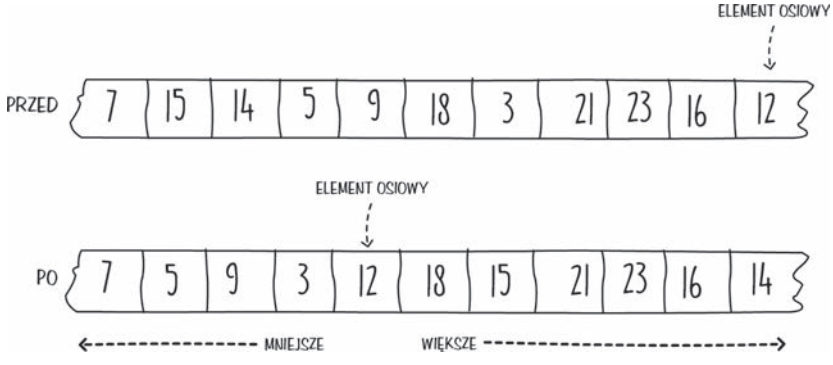
```
        return binarySearch(x, sortedNumbers, start, mid - 1);
        return binarySearch(x, sortedNumbers, mid + 1, end);
      }
    return false;}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Rekurencja jest niezbędnym narzędziem każdego programisty i będziemy z niej korzystać w tej książce wielokrotnie. W tym punkcie zaimplementowaliśmy przykład wyszukiwania binarnego. W następnym przyjrzymy się dokonywaniu podziału w algorytmie wyszukiwania szybkiego.

Podział w wyszukiwaniu szybkim

Podział jest procesem, w którym zmieniamy kolejność elementów tablicy tak, by elementy o wartości mniejszej niż element osiowy przenieść na jego lewą stronę, a elementy o wartości większej przesunąć na prawo (rysunek 2.2). Można to zrobić na wiele sposobów. Tutaj opiszę łatwy do zrozumienia schemat znany jako podział Lomuto.



Rysunek 2.2. Przed podziałem tablicy i po podziale tablicy

Istnieje wiele innych schematów. Schemat Lomuto ma tę wadę, że nie jest zbyt wydajny, gdy używa się go na już posortowanych listach. Oryginalny schemat podziału Hoare'a ma lepszą wydajność i przetwarza tablicę z obu końców. Działa lepiej, ponieważ dokonuje mniej zamian elementów, choć też jest mało efektywny, jeśli dane wejściowe są posortowane. Zarówno schemat Lomuto, jak i Hoare'a powodują, że sortowanie odbywa się w sposób niestabilny. Stabilność sortowania oznacza, że jeżeli co najmniej dwa elementy mają tę samą kluczową wartość, to pojawią się na wyjściu w tej samej kolejności, w której pojawiły się na wejściu. Istnieją inne schematy, sprawiające, że sortowanie szybkie jest stabilne, ale wykorzystują one więcej pamięci.

Aby dobrze zrozumieć ten schemat podziału, najlepiej uprościć działanie algorytm do pięciu prostych kroków:

1. Obierz skrajny prawy element tablicy za element osiowy.
2. Zacznij od lewej strony i znajdź element większy od elementu osiowego.
3. Zamień ten element miejscami z następnym, który jest mniejszy niż element osiowy.
4. Powtarzaj kroki 2. i 3., dopóki zamiana jest możliwa.
5. Pierwszy element, którego wartość jest większa od elementu osiowego, zamień miejscami z elementem osiowym.

Aby podział za pomocą tych kroków był efektywny, skorzystamy z dwóch wskaźników, z których jeden będzie wskazywał pierwszy element większy niż element osiowy, a drugi posłuży do szukania wartości mniejszej niż wartość elementu osiowego.

W kodzie z listingu 2.7 te wskaźniki do liczb całkowitych noszą nazwy, odpowiednio, `x` oraz `i`. Na początku algorytm wybiera jako element osiowy ostatni element tablicy wejściowej. Następnie za pomocą zmiennej `i` w pojedynczym przebiegu przetwarza tablicę od lewej do prawej. Jeśli element wskazywany obecnie przez `i` jest mniejszy niż element osiowy, to następuje inkrementacja zmiennej `x` oraz zamiana jej miejscami z `i`. Po zastosowaniu tej techniki zmienna `x` wskazuje element większy niż element osiowy albo ma tę samą wartość co zmienna `i`, a w tym przypadku zamiana elementów nie zmodyfikuje tablicy. Po zakończeniu pętli wykonujemy ostatni krok, zamieniając miejscami pierwszy element większy od elementu osiowego z samym elementem osiowym.

Listing 2.7. Podział w sortowaniu szybkim. Nazwa klasy źródłowej: QuickSort

```
private int partition(int[] numbers, int start, int end) {
  int pivot = numbers[end];
  int x = start - 1;
  for (int i = start; i < end; i++) {
    if (numbers[i] < pivot) {
      x++;
      swap(numbers, x, i);
    }
  }
  swap(numbers, x + 1, end);
  return x + 1;
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Ćwiczenie: Zrozumienie metody dzielącej

Scenariusz

Aby lepiej zrozumieć metodę dzielącą zastosowaną w listingu 2.7, przeanalizuj ją krok po kroku, korzystając z przykładowych danych.

Cel

Zrozumienie, jak przebiega podział Lomuto.

Kroki do wykonania

1. Uruchom w wyobraźni kod z listingu 2.7 dla każdego elementu tablicy, zwiększając wartości zmiennych `x` oraz `i`.
2. Wypełnij tabelę 2.1, przyjmując, że element osiowy jest ostatni na liście, czyli ma wartość 16.

Tabela 2.1. Kroki procesu podziału

i	tablica	x
-	[4, 5, 33, 17, 3, 21, 1, 16]	-1
0	[4, 5, 33, 17, 3, 21, 1, 16]	0
1		
2	[4, 5, 33, 17, 3, 21 1, 16]	1
3		
4	[4, 5, 3, 17, 33, 21, 1, 16]	2
5		
6		
7		
na koniec	[4, 5, 3, 1, 16, 21, 17, 33]	3

Teraz na pewno rozumiesz, jak przebiega podział w sortowaniu szybkim. W następnym punkcie włączymy metodę dzielącą do całego algorytmu sortowania szybkiego.

Jak to wszystko poskładać razem

Sortowanie szybkie pochodzi z algorytmów typu „dziel i zwyciężaj". W tej książce zobaczysz wiele innych przykładów algorytmów tej klasy, a metodą „dziel i zwyciężaj" zajmiemy się szczegółowo w rozdziale 4., „Paradygmaty projektowania algorytmów". Na razie ważne jest, by pamiętać, że algorytmy typu „dziel i zwyciężaj" dzielą problem na coraz mniejsze części, aż ich rozwiązanie stanie się banalne. Takie dzielenie można łatwo zaimplementować za pomocą rekurencji.

W przypadku sortowania szybkiego dzielimy rekurencyjnie tablicę, dopóki problem nie stanie się na tyle mały, że będzie się dało go łatwo rozwiązać. Gdy tablica ma tylko jeden element, rozwiązanie jest proste: tablica pozostaje niezmieniona, ponieważ nie ma w niej nic do sortowania. Jest to warunek zatrzymania algorytmu rekurencyjnego. Jeśli tablica ma więcej niż jeden element, to nadal ją dzielimy, korzystając z metody opracowanej w poprzednim punkcie.

> Istnieje również nierekurencyjny algorytm sortowania szybkiego, korzystający ze struktury danych stosu, ale jego napisanie jest nieco bardziej skomplikowane. Stosy i listy omówię w dalszej części tego rozdziału.

Listing 2.8 zawiera kompletny pseudokod sortowania szybkiego. Podobnie jak w większości funkcji rekurencyjnych, kod ten rozpoczyna się od sprawdzenia warunku zatrzymania. W tym przypadku sprawdzamy, czy tablica ma co najmniej dwa elementy, upewniając się, że początkowy wskaźnik tablicy jest mniejszy od końcowego.

Listing 2.8. Pseudokod rekurencyjnego sortowania szybkiego

```
quickSort(array, start, end)
  if(start < end)
    p = partition(array, start, end)
    quickSort(array, start, p - 1)
    quickSort(array, p + 1, end)
```

Jeśli tablica zawiera co najmniej dwa elementy, wywołujemy metodę dzielącą. Następnie, używając ostatniej pozycji elementu osiowego (zwróconej przez metodę dzielącą), przy użyciu sortowania szybkiego sortujemy rekurencyjnie lewą część, a potem prawą.

W tym celu wywołujemy ten sam kod sortowania szybkiego, korzystając ze wskaźników (`start`, `p – 1`) i (`p + 1`, `end`), bez uwzględnienia `p`, czyli pozycji elementu osiowego.

Aby zrozumieć działanie sortowania szybkiego, warto uświadomić sobie, że po wywołaniu metody dzielącej nie trzeba już przenosić elementu znajdującego się na zwróconej pozycji (elementu osiowego). Dzieje się tak dlatego, że wszystkie elementy po prawej stronie są większe, a po lewej mniejsze, więc element osiowy znajduje się w prawidłowej pozycji końcowej.

Implementacja sortowania szybkiego

Aby zaimplementować sortowanie szybkie w Javie, wykonaj następujące kroki:

1. Zaimplementuj w Javie pseudokod z listingu 2.8, wywołując metodę dzielącą pokazaną na listingu 2.7.
2. Rekurencyjna implementacja w Javie korzystająca z opracowanej w poprzednim punkcie metody dzielącej pokazana jest na listingu 2.9.

Listing 2.9. Sposób implementacji sortowania szybkiego. Nazwa klasy źródłowej: QuickSort

```
private void sort(int[] numbers, int start, int end) {
  if (start < end) {
    int p = partition(numbers, start, end);
      sort(numbers, start, p - 1);
      sort(numbers, p + 1, end);
  }
}
```

W tym punkcie opisałem algorytm sortowania szybkiego, znacznie wydajniejszy niż algorytm sortowania bąbelkowego, który pokazałem wcześniej. Przeciętnie algorytm ten wykonuje się w czasie $O(n \log n)$, co stanowi ogromną poprawę w stosunku do sortowania bąbelkowego, którego złożoność czasowa wynosi $O(n^2)$. Jednak w przypadku pesymistycznym algorytm sortowania szybkiego nadal działa w czasie $O(n^2)$. O tym, jakie dane wejściowe są najgorsze, decyduje zastosowany schemat podziału. W omówionym w tym podrozdziale schemacie Lomuto przypadek pesymistyczny zachodzi wtedy, gdy dane wejściowe są już posortowane. W następnym podrozdziale przeanalizujemy inny algorytm sortowania, którego czas wykonania w przypadku pesymistycznym wynosi $O(n \log n)$.

Korzystanie z sortowania przez scalanie

Chociaż sortowanie szybkie w przypadku oczekiwanym jest dość wydajne, wciąż ma teoretyczną pesymistyczną złożoność czasową $O(n^2)$. W tym podrozdziale poddamy analizie inny algorytm sortowania, zwany sortowaniem przez scalanie, którego pesymistyczna złożoność czasowa wynosi $O(n \log n)$. Podobnie jak sortowanie szybkie, sortowanie przez scalanie należy do algorytmów typu „dziel i zwyciężaj".

Sortowanie przez scalanie można streścić w trzech prostych krokach:

1. podziel tablicę na środku;
2. rekurencyjnie sortuj każdą część osobno;
3. połącz dwie posortowane części.

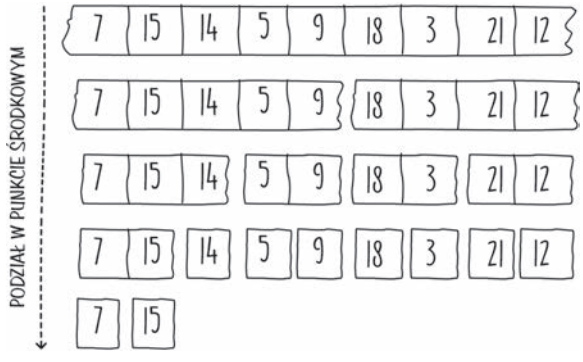
W następnym punkcie będziemy stopniowo opracowywać powyższe kroki, by coraz lepiej zrozumieć działanie sortowania przez scalanie.

> Chociaż sortowanie przez scalanie jest teoretycznie wydajniejsze niż sortowanie szybkie, w praktyce niektóre implementacje sortowania szybkiego są bardziej efektywne. Poza tym sortowanie przez scalanie ma złożoność pamięciową O(n), w przeciwieństwie do sortowania szybkiego, którego złożoność pamięciowa wynosi O(log n).

Dzielenie problemu

W poprzedniej sekcji pokazałem, jak użyć techniki rekurencyjnej, by podzielić problem na wiele mniejszych, aż staną się łatwe do rozwiązania. W sortowaniu przez scalanie korzysta się z tego samego podejścia. Przypadek bazowy naszej rekurencji jest taki sam jak w sortowaniu szybkim. Zachodzi wówczas, gdy tablica ma tylko jeden element, bo wtedy jest już posortowana.

Na rysunku 2.3 został pokazany podział tablicy w sortowaniu przez scalanie. W każdym kroku znajdujemy punkt środkowy tablicy i dzielimy ją na dwie części. Następnie rekurencyjnie sortujemy osobno lewą i prawą część rozdzielonej tablicy. Wywołanie rekurencyjne zatrzymujemy, gdy suma elementów do posortowania będzie równa wartości pokazanej na rysunku.



Rysunek 2.3. Etapy podziału w algorytmie sortowania przez scalanie

Implementacja sortowania przez scalanie

Uzupełnijmy pseudokod algorytmu sortowania przez scalanie.

Mając na uwadze, że rekurencyjna część sortowania przez scalanie jest bardzo podobna do algorytmu sortowania szybkiego z poprzedniego podrozdziału, uzupełnij pseudokod pokazany na listingu 2.10.

Listing 2.10. Ćwiczenie w uzupełnianiu pseudokodu rekurencyjnego sortowania przez scalanie

```
mergeSort(array, start, end)
  if(_____________)
    midPoint = _________
    mergeSort(array, _____, _____)
    mergeSort(array, _____, _____)
    merge(array, start, midPoint, end)
```

Pseudokod sortowania przez scalanie można uzupełnić w sposób pokazany na listingu 2.11.

Listing 2.11. Sposób uzupełnienia pseudokodu sortowania przez scalanie

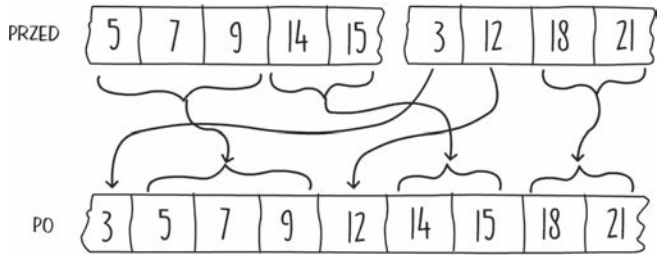
```
mergeSort(array, start, end)
  if(start < end)
```

```
midPoint = (end - start) / 2 + start
mergeSort(array, start, midPoint)
mergeSort(array, midPoint + 1, start)
merge(array, start, midPoint, end)
```

Algorytm sortowania przez scalanie należy do tej samej klasy algorytmów co sortowanie szybkie, jednak jego złożoność czasowa i pamięciowa jest inna. Zamiast dzielić tablicę względem elementu osiowego, sortowanie przez scalanie zawsze dzieli tablicę w punkcie środkowym. Jest to proces podobny do wyszukiwania binarnego, a jego wynikiem jest $\log_2 n$ podziałów tablic. W następnym punkcie przedstawię część scalającą tego algorytmu, łączącą dwa różne fragmenty podzielonej tablicy w jeden posortowany.

Scalanie problemu

Jak scalić dwie posortowane listy w jedną? Dokonuje tego funkcja `merge()`, znajdująca się na końcu pseudokodu pokazanego w poprzednim punkcie. Proces ten pokazałem na rysunku 2.4. Scalenie dwóch posortowanych list jest łatwiejsze niż sortowanie od podstaw.



Rysunek 2.4. Przed scaleniem i po scaleniu dwóch posortowanych tablic

Podobnie było w przypadku problemu znajdowania części wspólnej zbiorów, przedstawionego w rozdziale 1., „Algorytmy i ich złożoność".

Scalanie możemy wykonać w czasie liniowym, wykorzystując tylko dwa wskaźniki i pustą tablicę, co zostało pokazane na rysunku 2.4.

> Ponieważ oba fragmenty podzielonej tablicy są posortowane, łatwo jest je scalić. Szukając analogii, warto przypomnieć sobie, że problem znajdowania części wspólnej zbiorów przedstawiony w rozdziale 1., „Algorytmy i ich złożoność", stał się dużo łatwiejszy, gdy obie tablice wejściowe zostały posortowane. Podobny algorytm stosuje się także tutaj.

Pseudokod scalania jest pokazany na listingu 2.12. W kodzie tym funkcja `copyArray()` pobiera tablicę źródłową będącą pierwszym argumentem i kopiuje ją do tablicy docelowej, czyli do drugiego argumentu. Zmienna `start` wskazuje, w którym miejscu tablicy docelowej umieścić pierwszy element tablicy źródłowej.

Listing 2.12. Pseudokod scalania w sortowaniu przez scalanie

```
merge(array, start, middle, end)
  i = start
  j = middle + 1
  arrayTemp = initArrayOfSize(end - start + 1)
  for (k = 0 until end-start)
    if (i <= middle && (j > end || array[i] <= array[j]))
      arrayTemp[k] = array[i]
      i++
    else
      arrayTemp[k] = array[j]
      j++
  copyArray(arrayTemp, array, start)
```

W części scalającej sortowania przez scalanie tworzymy tymczasową tablicę o wielkości równej łącznemu rozmiarowi dwóch fragmentów tablicy początkowej. Następnie wykonujemy pojedyncze przejście po tej tablicy, wypełniając każdy kolejny element kolejną najmniejszą wartością z dwóch list wejściowych (reprezentowanych przez wskaźniki `start`, `middle` i `end`). Po wybraniu elementu z jednej z list przesuwamy wskaźnik tej listy i powtarzamy to działanie aż do zakończenia scalania.

> W języku Java istnieje wiele narzędzi, których można użyć do zaimplementowania funkcji `copyArray()` pokazanej na końcu listingu 2.12. Możemy samodzielnie zaimplementować funkcję `copy()`, korzystając z pętli `for`. Można też użyć strumieni Javy i zapisać kopię tablicy w jednym wierszu kodu. Bodaj najprostszym sposobem jest skorzystanie z funkcji `System.arrayCopy()`.

Sortowanie przez scalanie to teoretycznie jeden z najszybszych algorytmów sortowania. Niestety zużywa ono przez to nieco więcej pamięci, chociaż istnieją implementacje, które w celu zaoszczędzenia pamięci dokonują scalania w miejscu.

Dla porównania w tabeli 2.2 przedstawiam kilka technik sortowania oraz ich złożoność czasową i pamięciową.

Tabela 2.2. Algorytmy sortowania

Algorytm sortowania	Przypadek przeciętny	Przypadek pesymistyczny	Pamięć	Stabilność
bąbelkowe	$O(n^2)$	$O(n^2)$	$O(1)$	stabilne
przez wybieranie	$O(n^2)$	$O(n^2)$	$O(1)$	niestabilne
przez wstawianie	$O(n^2)$	$O(n^2)$	$O(1)$	stabilne
szybkie	$O(n \log n)$	$O(n^2)$	$O(1)$	niestabilne
przez scalanie	$O(n \log n)$	$O(n \log n)$	$O(n)$	stabilne
przez kopcowanie	$O(n \log n)$	$O(n \log n)$	$O(1)$	niestabilne

Ćwiczenie: Implementacja sortowania przez scalanie w języku Java

Scenariusz

Sortowanie przez scalanie jest jedną z najszybszych technik sortowania. Korzysta z niego wiele dołączanych bibliotek i interfejsów API. W tym ćwiczeniu napiszemy w Javie algorytm sortujący w ten sposób tablicę.

Cel

Wykorzystanie pseudokodu pokazanego w tym punkcie do implementacji pełnego algorytmu sortowania przez scalanie w języku Java.

Warunki wstępne

Aby wykonać to ćwiczenie, zaimplementuj metody znajdujące się w klasie dostępnej w repozytorium dołączonym do tej książki pod adresem *src/main/java/com/packt/datastructuresandalg/lesson2/activity/mergesort/MergeSort.java*

Jeśli skonfigurowałeś projekt, testy jednostkowe do tego ćwiczenia uruchomisz następującym poleceniem:

```
gradlew test --tests com.packt.datastructuresandalg.lesson2.activity.mergesort*
```

Kroki do wykonania

1. Rozpocznij od implementacji metody `mergeSort`, która dzieli tablicę na dwie części, sortuje je obie rekurencyjnie i scala wyniki.
2. Następnie zaimplementuj metodę `merge`, scalającą oba fragmenty podzielonej tablicy w innym miejscu.
3. Po zakończeniu scalania skopiuj nową tablicę z powrotem do tablicy wejściowej.

Rozpoczęcie pracy z podstawowymi strukturami danych

Struktury danych pozwalają uporządkować dane w taki sposób, by były efektywnie dostępne podczas rozwiązywania problemu. Wybór odpowiedniej struktury danych zależy od typu problemu do rozwiązania (który określa sposób dostępu do danych), ilości danych, które należy uporządkować, oraz nośnika używanego do przechowywania danych (pamięć, dysk i tak dalej).

Miałeś już okazję zobaczyć jedną ze struktur danych i jej używać. W poprzednich podrozdziałach często korzystaliśmy z tablic. Są one najprostszymi strukturami. Zapewniają dostęp do danych za pomocą indeksu i mają ustalony rozmiar (bywają nazywane również strukturami statycznymi). Stanowią przeciwieństwo innych, dynamicznych struktur danych, które można powiększać, zapewniając więcej miejsca na dane, kiedy tylko zajdzie taka potrzeba.

Wprowadzenie do struktur danych

Formalnie rzecz biorąc, struktura danych to organizacja elementów danych — zbiór funkcji, które można zastosować do tych danych (takich jak dodawanie, usuwanie i wyszukiwanie), oraz wszelkie powiązania między różnymi elementami. Tabela 2.3 przedstawia typowe operacje wykonywane na strukturach danych.

Tabela 2.3. Niektóre typowe operacje na strukturach danych

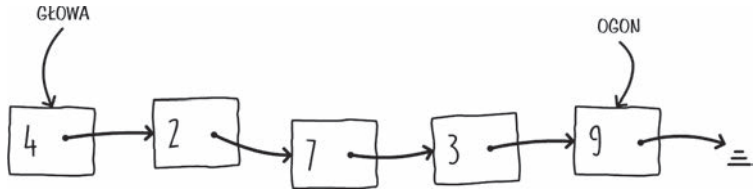
Operacja	Typ	Opis
`search(klucz)`	niemodyfikująca	Operacja, która po podaniu klucza do określonej wartości zwraca tę wartość, jeśli można ją znaleźć w strukturze
`size()`	niemodyfikująca	Całkowita liczba wartości przechowywanych w strukturze danych
`add(wartość)`	modyfikująca	Wstawia wartość do struktury danych
`update(klucz, wartość)`	modyfikująca	Aktualizuje istniejącą pozycję, korzystając z podanego klucza i wartości
`delete(wartość)`	modyfikująca	Usuwa element ze struktury danych
`minimum()`	niemodyfikująca	Operacja obsługiwana tylko przez uporządkowane struktury danych, zwracająca wartość najmniejszego klucza
`maximum()`	niemodyfikująca	Operacja obsługiwana tylko przez uporządkowane struktury danych, zwracająca wartość największego klucza

W tym podrozdziale zobaczysz różnego typu dynamiczne struktury danych. Zaczniemy od list powiązanych, które są zoptymalizowane pod kątem dynamicznego zwiększania wielkości, ale wyszukiwanie w nich jest powolne. Potem na bazie list zaimplementujemy inne struktury danych, takie jak kolejki i stosy.

Struktura list powiązanych

Lista powiązana to lista elementów, z których każdy zawiera jedynie informacje o następnym elemencie listy, jeśli taki istnieje. Rysunek 2.5 pokazuje przykład takiej listy. Każde pole na rysunku reprezentuje pojemnik na element danych, który mamy przechowywać. Ten pojemnik, zwany węzłem, zawiera wartości naszych danych i wskaźnik do następnego węzła listy. Jak widać na rysunku, węzeł z początku listy jest nazywany głową (ang. *head*), a ostatni element listy — ogonem (ang. *tail*).

Dla ułatwienia dostępu do tych węzłów struktura danych przechowuje do nich oddzielne wskaźniki.



Rysunek 2.5. Przykład listy powiązanej

Zaletą korzystania z listy powiązanej, w przeciwieństwie do tablicy, jest możliwość dynamicznego zwiększania wielkości. Przy korzystaniu z tablicy miejsce alokuje się na początku i pozostaje ono stałe. Przydzielenie na nią zbyt dużo miejsca, które pozostanie niewykorzystane, to marnotrawstwo zasobów. Z drugiej strony, jeśli tablica jest zbyt mała, dane mogą się w niej nie zmieścić. Natomiast w przypadku listy powiązanej miejsce nie jest ustalone. Struktura ta powiększa się dynamicznie w miarę dodawania danych i zmniejsza się podczas ich usuwania, zwalniając miejsce w pamięci.

Używając języka obiektowego, takiego jak Java, stwórzmy model listy powiązanej z wykorzystaniem oddzielnych, połączonych ze sobą instancji węzłów. Listing 2.13 przedstawia, jak odwzorować węzeł listy w klasie Javy.

Listing 2.13. Klasa węzła listy powiązanej (dla zwięzłości pominięto akcesory i mutatory). Nazwa klasy źródłowej: Linkedlistnode

```
public class LinkedListNode<V> {
  private V value;
  private LinkedListNode<V> next;
  public LinkedListNode(V value, LinkedListNode<V> next) {
    this.value = value;
    this.next = next;
  }
  public Optional<LinkedListNode<V>> getNext() {
    return Optional.ofNullable(next);
  }
}
```

Klasa ta zawiera samoreferencję, co pozwala połączyć wiele węzłów w listę, jak pokazałem w rysunku 2.5.

> Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Zwróć uwagę na to, że do wskazania następnego węzła używamy obiektów klasy `Optional` języka Java, co pozwala uniknąć zwracania pustych wskaźników. W węźle końcowym listy powiązanej obiekt taki jest zawsze pusty. Do modelowania przechowywanych danych wykorzystujemy z kolei typy generyczne. W ten sposób zachowujemy strukturę tak ogólną, jak to tylko możliwe, przechowującą dane dowolnego typu.

Klasa `Optional`, wprowadzona w 8. wersji języka Java do reprezentacji wartości opcjonalnych, pozwala uniknąć używania wartości `null`.

Przekształcenie listy powiązanej na listę powiązaną dwukierunkowo

W języku Java zmodyfikujemy klasę węzła w celu obsługi listy dwukierunkowej.

Lista dwukierunkowa to lista, w której każdy węzeł zawiera odniesienia do następnego i poprzedniego węzła. Spróbuj zmodyfikować kod z listingu 2.13 tak, by obsługiwał listę dwukierunkową.

Rozwiązanie przedstawia listing 2.14.

Listing 2.14. Klasa węzła listy dwukierunkowej (dla zwięzłości pominięto akcesory i mutatory). Nazwa klasy źródłowej: Dbllinkedlistnode

```
public class DblLinkedListNode<V> {
  private V value;
  private DblLinkedListNode<V> next;
  private DblLinkedListNode<V> previous;
  public DblLinkedListNode(V value,
  DblLinkedListNode<V> next,
  DblLinkedListNode<V> previous) {
    this.value = value;
    this.next = next;
    this.previous = previous;
  }
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*. W liście dwukierunkowej węzeł początkowy ma pusty wskaźnik do poprzednika, a węzeł końcowy ma pusty wskaźnik do następnika.

W tym podpunkcie pokazałem, jak modelować węzeł listy powiązanej za pomocą klas, typów generycznych i obiektów `Optional`. W następnym podpunkcie zobaczysz, jak zaimplementować operacje na liście.

Operacje na listach powiązanych

Zanim będziesz mógł dokonywać operacji na liście powiązanej, musisz zainicjalizować tę strukturę danych i oznaczyć ją jako pustą. Koncepcyjnie jest to sytuacja, gdy wskaźnik do początku listy nic nie wskazuje. Dodamy tę logikę do konstruktora w języku Java.

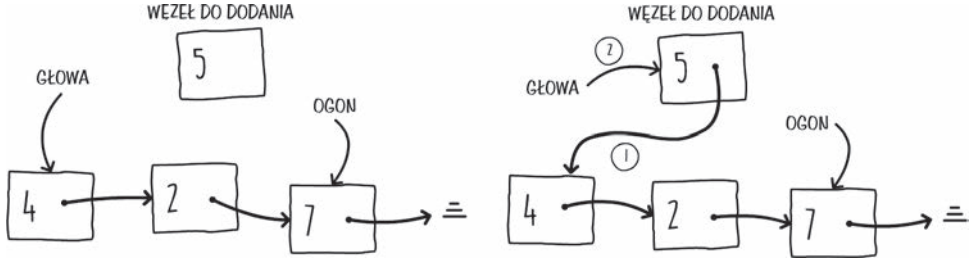
Pokazuje to listing 2.15. Zauważ, że po raz kolejny w celu przechowania w liście elementów dowolnego typu korzystamy z typów generycznych:

Listing 2.15. Inicjalizowanie listy powiązanej za pomocą konstruktora. Nazwa klasy źródłowej: Linkedlist

```
public class LinkedList<V> {
  private LinkedListNode<V> head;
  public LinkedList() {
    head = null;
  }
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Jak dodawać elementy na początku listy i usuwać je stamtąd? Dodanie węzła do listy powiązanej wymaga ponownego przypisania dwóch wskaźników. W nowym węźle wskaźnikowi do następnika przypisuje się tę samą wartość, którą ma wskaźnik do głowy. Potem wskaźnik do głowy należy ustawić tak, by wskazywał nowo utworzony węzeł. Proces ten został pokazany na rysunku 2.6. Usunięcie elementu z początku listy to proces odwrotny. Wskaźnik do głowy należy ustawić na następnik dotychczasowego węzła początkowego. Dla porządku w węźle początkowym wskaźnikowi do następnika można nadać wartość pustą.



Rysunek 2.6. Dodawanie węzła na początku listy

Aby zlokalizować element na liście, przechodzimy przez nią, dopóki nie znajdziemy poszukiwanego elementu lub nie dojdziemy do jej końca. Możemy tego dokonać w prosty sposób, zaczynając od początku listy i podążając ciągle za wskaźnikiem do następnego węzła, aż znajdziemy węzeł z poszukiwaną wartością, albo ich zabraknie, czyli wskaźnik do następnego węzła będzie pusty.

Na listingu 2.16 pokazałem operacje dodawania węzła na początku listy powiązanej [`addFront()`] i usuwania węzła z początku listy [`deleteFront()`]. W przypadku metody `addFront()` tworzymy nowy węzeł, a jego wskaźnikowi do następnika przypisujemy bieżącą wartość wskaźnika do głowy. Następnie ustawiamy nowy węzeł jako głowę. Zwróć uwagę, że w metodzie usuwającej korzystamy z obiektów `Optional` języka Java. Jeśli wskaźnik do głowy ma wartość `null`, takim pozostaje i nic nie zmieniamy. W przeciwnym razie przekształcamy go na pierwszy wskaźnik do następnika. Wreszcie w węźle początkowym wskaźnikowi do następnika nadajemy wartość `null`. Ten ostatni krok nie jest konieczny, ponieważ osierocony węzeł zostanie zebrany podczas odśmiecania, jednak uwzględnimy go w celu zachowania kompletności.

Listing 2.16. Dodawanie i usuwanie na początku listy powiązanej. Nazwa klasy źródłowej: Linkedlist

```
public void addFront(V item) {
  this.head = new LinkedListNode<>(item, head);
}
public void deleteFront() {
  Optional<LinkedListNode<V>> firstNode = Optional.
  ofNullable(this.head);
  this.head = firstNode.flatMap(LinkedListNode::getNext).
  orElse(null);
  firstNode.ifPresent(n -> n.setNext(null));
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Na listingu 2.17 pokazałem jeden ze sposobów implementacji metody wyszukującej. Tu znów można zaobserwować, jak korzystać w Javie z metod klasy `Optional`. Pętlę `while` rozpoczynamy od wskaźnika do głowy i jeśli bieżący węzeł nie zawiera poszukiwanego elementu, przechodzimy dalej do następnego węzła, o ile taki istnieje. Następnie zwracamy ostatni wskaźnik, w którym może być pusty obiekt klasy `Optional` albo węzeł zawierający dopasowanie.

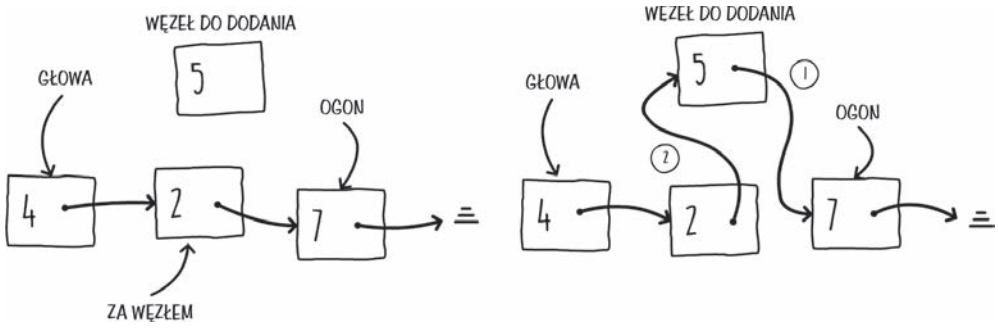
Listing 2.17. Dodawanie i usuwanie na początku listy powiązanej. Nazwa klasy źródłowej: Linkedlist

```
public Optional<LinkedListNode<V>> find(V item) {
  Optional<LinkedListNode<V>> node = Optional.ofNullable(this.head);
  while (node.filter(n -> n.getValue() != item).isPresent()) {
    node = node.flatMap(LinkedListNode::getNext);
  }
  return node;
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*. Metoda `find()` na liście powiązanej ma pesymistyczną złożoność czasową rzędu $O(n)$. Dzieje się tak, gdy pasujący element jest na końcu listy lub nie ma go w niej w ogóle.

W poprzednim przykładzie pokazałem, jak dodać element na początku listy powiązanej. A jak wstawić go do listy w dowolnie wybranym punkcie? Rysunek 2.7 przedstawia, jak to zrobić w dwóch krokach.

Jak to zrobić, pokazuje listing 2.18. Jest to metoda Javy o nazwie `addAfter()`, pobierająca węzeł i element do wstawienia. Metoda ta dodaje węzeł zawierający element po węźle podanym w argumencie `aNode`. Implementacja przebiega zgodnie z krokami pokazanymi na rysunku 2.7.



Rysunek 2.7. Dodawanie węzła na dowolnej pozycji listy

Listing 2.18. Przykładowa metoda addAfter. Nazwa klasy źródłowej: Linkedlist

```
public void addAfter(LinkedListNode<V> aNode, V item) {
  aNode.setNext(new LinkedListNode<>(item, aNode.getNext().orElse(null)));
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Ćwiczenie: Przechodzenie przez listę powiązaną

Scenariusz

Mając listę powiązaną, która zawiera kilka elementów, mamy zbudować łańcuch o postaci `[3,6,4,2,4]`. Jeśli lista jest pusta, należy wypisać `[]`.

Cel

Napisać w Javie kod przechodzący przez listę powiązaną.

Kroki do wykonania

1. Do klasy `LinkedList` dopisz metodę `toString()`:

```
public String toString() {
}
```

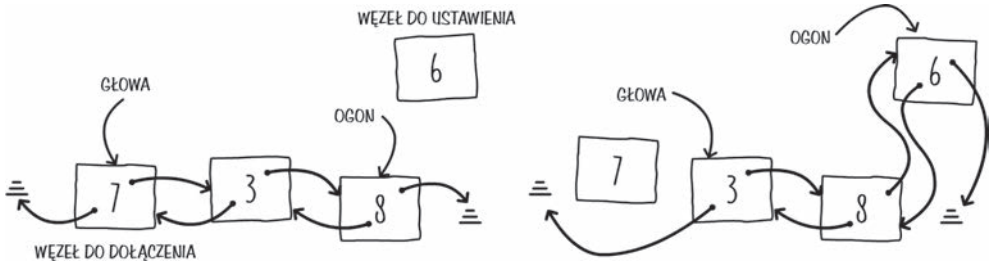
2. W pętli `while` przejdź przez listę powiązaną.

W tym punkcie pokazałem, jak zaimplementować różne operacje na liście powiązanej. Ta struktura danych będzie podstawą, którą wykorzystamy do modelowania kolejek i stosów. Listy powiązane będą też często stosowane w bardziej zaawansowanych algorytmach w dalszej części książki.

Kolejki

Kolejki są abstrakcyjnymi strukturami danych mającymi naśladować działanie prawdziwych kolejek. Mają różnorodne zastosowania, między innymi przydzielanie zasobów, planowanie, sortowanie i wiele innych Zazwyczaj implementuje się je przy użyciu listy dwukierunkowej, chociaż istnieją inne sposoby. Kolejka umożliwia zwykle dwie operacje: operację **ustawiania** (ang. *enqueue*), czyli dodawania elementów na końcu kolejki, oraz przeciwstawną operację **odłączania** (ang. *dequeue*), w której elementy są usuwane z początku kolejki. Na tych dwóch operacjach opiera się zasada działania tej struktury danych, określana po angielsku jako ***First In First Out*** (**FIFO** — z ang. pierwszy na wejściu, pierwszy na wyjściu).

Kolejkę można efektywnie zaimplementować przy użyciu listy dwukierunkowej. Implementacja **odłączania z kolejki** polega wtedy na usunięciu elementu z początku listy powiązanej. **Ustawienie w kolejce** to dodanie elementu na koniec listy. Rysunek 2.8 pokazuje, jak wykonać te dwie operacje.



Rysunek 2.8. Ustawianie w kolejce i odłączanie z niej przy użyciu listy dwukierunkowej

Aby odłączyć element z kolejki opartej na liście dwukierunkowej, wystarczy przenieść jej głowę do następnego elementu na liście i odłączyć dotychczasowy element początkowy, nadając jego wskaźnikowi do poprzednika wartość pustą. Ustawianie elementu na końcu listy jest procesem trzyetapowym. W nowym węźle należy ustawić wskaźnik do poprzednika na bieżący element końcowy, potem w dotychczasowym ogonie trzeba ustawić wskaźnik do następnika na nowy węzeł, a na końcu przestawić wskaźnik do ogona na nowy węzeł. Pseudokod obu tych operacji pokazany jest na listingu 2.19.

Listing 2.19. Ustawianie w kolejce i odłączanie z niej przy użyciu listy dwukierunkowej

```
dequeue(head)
  if (head != null)
    node = head
    head = head.next
    if (head != null) head.previous = null
    return node.value
  return null
enqueue(tail, item)
  node = new Node(item)
  node.previous = tail
```

```
if (tail != null) tail.next = node
if (head == null) head = node
tail = node
```

Dodawanie i usuwanie elementów w kolejce

Aby zaimplementować w Javie metody `enqueue()` i `dequeue()`, wykonaj następujące kroki:

1. Korzystając z listy dwukierunkowej, zaimplementuj w Javie pseudokod odłączania i ustawiania w kolejce z poprzedniego listingu. Zastosuj strukturę klasy i sygnatury metod pokazane na listingu 2.20.

Listing 2.20. Struktura klasy ćwiczeniowej i sygnatury metod

```
public class Queue<V> {
  private DblLinkedListNode<V> head;
  private DblLinkedListNode<V> tail;
  public void enqueue(V item)
  public Optional<V> dequeue()
}
```

2. Metodę `enqueue()` można zaimplementować tak jak na listingu 2.21.

Listing 2.21. Struktura klasy ćwiczeniowej i sygnatury metod. Nazwa klasy źródłowej: Queue

```
public void enqueue(V item) {
  DblLinkedListNode<V> node = new DblLinkedListNode<>(item, null,
tail);
  Optional.ofNullable(tail).ifPresent(n -> n.setNext(node));
  tail = node;
  if(head == null) head = node;
}
```

> Kod metody `dequeue()` znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Kolejki są dynamicznymi strukturami danych uporządkowanymi według zasady FIFO. W następnym punkcie zbadamy strukturę danych o innym uporządkowaniu zwaną stosem.

Stosy

Stosy, choć też zazwyczaj implementowane przy użyciu list powiązanych, działają inaczej niż kolejki. Nie są one uporządkowane według zasady FIFO, lecz **LIFO — *Last In First Out*** (z ang. ostatni na wejściu, pierwszy na wyjściu; patrz: rysunek 2.9). Wykonuje się na nich dwie

Rysunek 2.9. Operacje odkładania i zdejmowania na stosie kartek papieru

główne operacje: **odkładanie** (ang. *push*), które oznacza dodanie elementu na wierzch stosu, i **zdejmowanie** (ang. *pop*), które polega na usunięciu i zwróceniu jednego elementu z wierzchu stosu. Podobnie jak kolejek, stosów używa się często w wielu algorytmach, takich jak przeszukiwanie w głąb, obliczanie wartości wyrażeń oraz wielu innych.

Aby odwzorować stos, wystarczy użyć prostej listy powiązanej. Do jej głowy można się odnosić jako do wierzchołka stosu. Do odkładania elementów na wierzch stosu można użyć opracowanej w poprzednich punktach metody `addFront()`. Operacja zdejmowania różni się tylko tym, że zwraca element typu `Optional` z wierzchołka stosu. Implementację metod `push` i `pop` w Javie widać na listingu 2.22. Zwróć uwagę, że operacja zdjęcia zwraca wartość typu `Optional`, która jest wypełniana wartością, jeśli stos nie jest pusty.

Do odwzorowania stosu wystarczy lista jednokierunkowa, ponieważ operujemy tylko na jednym z jej końców. W przypadku kolejki musieliśmy modyfikować zarówno głowę, jak i ogon listy powiązanej, dlatego lepiej było użyć listy dwukierunkowej. Implementację metod `push()` i `pop()` przedstawia listing 2.22.

Listing 2.22. Operacje odkładania i zdejmowania w Javie. Nazwa klasy źródłowej: Stack

```
public void push(V item) {
  head = new LinkedListNode<V>(item, head);
}
public Optional<V> pop() {
  Optional<LinkedListNode<V>> node = Optional.ofNullable(head);
  head = node.flatMap(LinkedListNode::getNext).orElse(null);
  return node.map(LinkedListNode::getValue);
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Odwracanie łańcucha znaków

Wykorzystamy strukturę danych stosu do odwrócenia łańcucha znaków.

Wykonaj następujące czynności:

1. Aby odwrócić łańcuch, odłóż na stos wszystkie znaki z wejścia, a następnie zdejmij je po jednym, tworząc odwrócony łańcuch. Sygnatura tej metody może wyglądać następująco:

```
public String reverse(String str)
```

2. Listing 2.23 pokazuje, jak odwrócić łańcuch przy użyciu struktury danych stosu.

Listing 2.23. Sposób odwracania łańcucha. Nazwa klasy źródłowej: StringReverse

```
public String reverse(String str) {
  StringBuilder result = new StringBuilder();
  Stack<Character> stack = new Stack<>();
  for (char c : str.toCharArray())
  stack.push(c);
  Optional<Character> optChar = stack.pop();
  while (optChar.isPresent()) {
    result.append(optChar.get());
    optChar = stack.pop();
  }
  return result.toString();
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Stosy są powszechnie stosowane w wielu algorytmach z dziedziny informatyki. W tym punkcie pokazałem, jak je zaimplementować w sposób dynamiczny, korzystając z list powiązanych. W następnym punkcie zobaczysz, jak modelować stosy i kolejki w sposób statyczny przy użyciu tablic.

Modelowanie stosów i kolejek przy użyciu tablic

Stosy i kolejki niekoniecznie muszą być dynamiczne. Jeśli wymagane dane mają stały rozmiar, wystarczy bardziej zwarta implementacja. Użycie tablic do modelowania stosów i kolejek gwarantuje, że struktura danych zwiększy się tylko do pewnej wielkości. Kolejną zaletą podejścia tablicowego, jeśli nie przeszkadzają nam ograniczenia statycznej struktury danych, jest lepsza wydajność pamięciowa. Problem ze statycznymi strukturami danych polega na tym, że kolejka lub stos może powiększyć się tylko do maksymalnego ustalonego rozmiaru początkowo przydzielonej tablicy.

Implementacja stosu przy użyciu tablicy wiąże się z zainicjalizowaniem pustej tablicy o stałym rozmiarze. Potem wystarczy pilnować, by wskaźnik w postaci indeksu, który początkowo ma wartość 0, wskazywał na wierzchołek stosu. Przy odkładaniu na stos umieszczamy w tym indeksie tablicy element i zwiększamy wskaźnik o 1. Podczas zdejmowania elementu zmniejszamy ten wskaźnik o 1 i odczytujemy wartość. Proces ten jest przedstawiony na listingu 2.24.

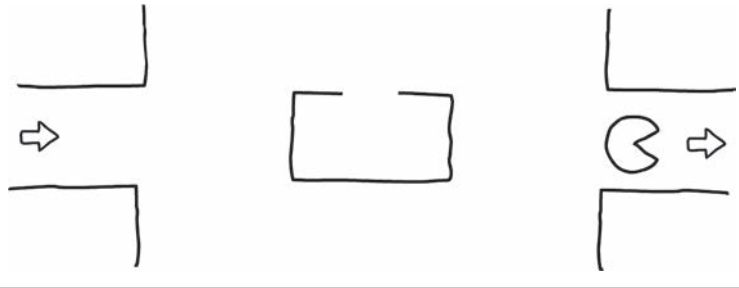
Listing 2.24. Stos z wykorzystaniem tablicy zamiast listy powiązanej. Nazwa klasy źródłowej: Stackarray

```
public StackArray(int capacity) {
  array = (V[]) new Object[capacity];
}
public void push(V item) {
  array[headPtr++] = item;
}
public Optional<V> pop() {
  if (headPtr > 0) return Optional.of(array[--headPtr]);
  else return Optional.empty();
}
```

Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Trochę więcej namysłu wymaga tablicowa implementacja kolejki. Trudność z kolejką polega na tym, że struktura ta jest modyfikowana z obu końców, ponieważ powiększa się od strony ogona, a zmniejsza od głowy.

Gdy ustawiamy w kolejce wartości i je z niej odłączamy, wydaje się, że kolejka przesuwa się w prawą stronę tablicy. Musimy poradzić sobie z tym, co się stanie, gdy zawartość kolejki dotrze do końca tablicy. Aby to działało, wystarczy, że pozwolimy danym przewijać się przez końce tablicy (spójrz na rysunek 2.10, aby przypomnieć sobie grę pacman):



Rysunek 2.10. Analogia przewijania się danych

Kiedy dojdziemy do końca tablicy, zaczniemy po prostu od jej początku. Ten mechanizm przewijania nazywa się **buforem cyklicznym** (ang. *circular buffer*). Implementuje się go za pomocą operatora modulo, aby uzyskać dostęp do dowolnego elementu bazowej tablicy, jak pokazałem na listingu 2.25. Zauważ, że po ustawieniu elementu w kolejce umieszczamy go w pozycji

końcowej i inkrementujemy wskaźnik do ogona kolejki za pomocą operatora `mod`. Gdy wskaźnik jest większy lub równy rozmiarowi tablicy, przewija się i zaczyna ponownie od zera. To samo dzieje się w przypadku metody `dequeue`, w której w podobny sposób uzyskujemy dostęp do wskaźnika do głowy i go inkrementujemy.

> Kod z listingu 2.25 przed wstawieniem do kolejki kolejnego elementu nie sprawdza, czy bufor cykliczny jest pełny. Implementacja tej kontroli będzie ćwiczeniem w następnym podpunkcie.

Listing 2.25. Wstawianie do kolejki i odłączanie z niej przy użyciu tablicy. Nazwa klasy źródłowej: QueueArray

```
public void enqueue(V item) {
  array[tailPtr] = item;
  tailPtr = (tailPtr + 1) % array.length;
}
public Optional<V> dequeue() {
  if (headPtr != tailPtr) {
    Optional<V> item = Optional.of(array[headPtr]);
    headPtr = (headPtr + 1) % array.length;
    return item;
  } else return Optional.empty();
}
```

> Kod tego przykładu znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Bezpieczne wstawianie do kolejki zaimplementowanej w tablicy

Napiszemy bezpieczną metodę `enqueue()`, która zakończy się niepowodzeniem, jeśli kolejka będzie pełna.

Wykonaj następujące kroki:

1. Zmodyfikuj metody `enqueue` i `dequeue` przedstawione na poprzednim listingu, tak by metoda `enqueue` zwracała wartość logiczną `false`, gdy kolejka jest pełna i nie może przyjąć kolejnych elementów.
2. Zaimplementuj metody o następujących sygnaturach:

   ```
   public boolean enqueueSafe(V item)
   public Optional<V> dequeueSafe()
   ```

3. Listing 2.26 przedstawia implementację metody `enqueueSafe()`, która zwraca wartość logiczną, jeśli kolejka jest pełna.

Listing 2.26. Bezpieczny sposób wstawiania do kolejki i odłączania z niej. Nazwa klasy źródłowej: QueueArray

```
private boolean full = false;
public boolean enqueueSafe(V item) {
```

```
    if (!full) {
      array[tailPtr] = item;
      tailPtr = (tailPtr + 1) % array.length;
      this.full = tailPtr == headPtr;
      return true;
    }
    return false;
  }
```

Kod metody `dequeueSafe()` znajdziesz w archiwum, które możesz pobrać pod adresem *ftp://ftp.helion.pl/przyklady/sdalgj.zip*.

Zamiast dynamicznej listy powiązanej zaimplementowaliśmy kolejki i stosy, stosując statyczną strukturę tablicy. Ma to tę zaletę, że na każdy element przypada mniej pamięci, ponieważ lista powiązana musi przechowywać wskaźniki do innych węzłów. Jednak dzieje się to kosztem ograniczenia wielkości struktury.

Ćwiczenie: Obliczanie wartości wyrażenia postfiksowego

Scenariusz

Jesteśmy przyzwyczajeni do pisania wyrażeń matematycznych w postaci 1+2·3. Ten rodzaj zapisu nazywa się **infiksowym**. Operator znajduje się w nim zawsze pomiędzy dwoma operandami. Istnieje inna notacja, zwana **postfiksową**, w której operator występuje po operandach. Przykłady takich wyrażeń są przedstawione w tabeli:

Wyrażenie infiksowe	Wyrażenie postfiksowe
1+2	1 2 +
1+2·3	1 2 3 · +
(1+2)·3	1 2 + 3 ·
5+4/2·3	5 4 2 / 3 · +

Cel

Zaimplementuj algorytm, który przyjmuje łańcuch wyrażenia postfiksowego, oblicza jego wartość i zwraca wynik.

Warunki wstępne

- Zaimplementuj w klasie dostępnej w dołączonym do tej książki repozytorium pod adresem *src/main/java/com/packt/datastructuresandalg/lesson2/activity/postfix/EvalPostfix.java* następującą metodę:

```
public double evaluate(String postfix)
```

- Załóż, że operator i operandy zawsze oddziela spacja, według wzoru „5 2 +”. Łańcuch wejściowy będzie wyglądał podobnie jak przykłady przedstawione w poprzedniej tabeli.

Jeśli skonfigurowałeś projekt, testy jednostkowe do tego ćwiczenia uruchomisz następującym poleceniem:

```
gradlew test --tests com.packt.datastructuresandalg.lesson2.activity.postfix*
```

Rozwiązanie stanie się o wiele prostsze, jeśli użyjesz jednej ze struktur danych, które badaliśmy w tym podrozdziale.

Kroki do wykonania

1. Aby rozwiązać ten problem, użyj struktury danych stosu.
2. Rozpocznij przetwarzanie wyrażenia od lewej do prawej.
3. Jeśli napotkasz operand liczbowy, odłóż go na stos.
4. Jeśli napotkasz operator, zdejmij dwa elementy ze stosu, wykonaj odpowiednią operację (dodawanie, odejmowanie i tak dalej) i odłóż wynik z powrotem na stos.
5. Po przetworzeniu całego wyrażenia na wierzchu stosu powinien znajdować się wynik.

Podsumowanie

W tym rozdziale położyliśmy podwaliny pod bardziej skomplikowane treści przedstawione na dalszych stronach książki. W pierwszych kilku podrozdziałach pokazałem, że prosty problem, taki jak sortowanie, może mieć wiele rozwiązań, z których każde ma inną charakterystykę wydajności. Zbadaliśmy trzy główne implementacje sortowania, to znaczy bąbelkowe, szybkie i przez scalanie.

W dalszych punktach zapoznałeś się ze strukturami danych i przeanalizowaliśmy różne implementacje i przypadki użycia list powiązanych, kolejek i stosów. Pokazałem też, jak użyć pewnych struktur danych jako elementów konstrukcyjnych do budowania bardziej złożonych struktur. W następnym rozdziale przestudiujemy dwie ważne i powszechnie stosowane struktury danych: tablice z haszowaniem i drzewa binarne.

Skorowidz

A

adresowanie
  kwadratowe, 72
  liniowe, 71, 73
  otwarte, 77
algorytm, 13
  A*, 160
  Aho-Corasick, 132
  Boyera-Moore'a, 120, 122
    implementacja, 129
  Dijkstry, 150
  Floyda-Warshalla, 154
  Knutha-Morrisa-Pratta, 132
  konwersji liczb, 14
  Rabina-Karpa, 130
  typu dziel i zwyciężaj, 103
  wyszukiwania naiwnego, 119
    implementacja, 120
    usprawnienie, 121
algorytmy
  identyfikacja złożoności, 26
  mierzenie złożoności, 16
  paradygmaty projektowania, 93
  zachłanne, 94
    implementacja, 102
    składniki, 96
  sortowania, 35
  wyszukiwania wzorca, 119–122, 130

B

binarne drzewo poszukiwań, 65, 80
  przechodzenie, 84
  zrównoważone, 86
bufor cykliczny, 61

C

cykle, 147

D

drzewo binarne, 78
  operacje, 80
  przechodzenie, 84
  rotacja, 89
  wyszukiwanie klucza, 83
dyskretny problem plecakowy, 111
dziel i zwyciężaj, 103

F

FIFO, First In First Out, 57
funkcje haszujące, 74

G

graf, 135
  nieskierowany, 136
  skierowany, 136
grafy
  listy sąsiedztwa, 137
  macierz sąsiedztwa, 139
  minimalne drzewa rozpinające, 159
  problem maksymalnego przepływu, 161
  przechodzenie, 142
  przeszukiwanie
    w głąb, 144
    wszerz, 142
  ścieżki, 149
  wykrywanie cykli, 147
grupowanie, 72

H

haszowanie, 66
  dwukrotne, 72
  modularne, 74
  przez mnożenie, 75
  uniwersalne, 76

I

implementacja
  adresowania otwartego, 77
  algorytmu
    Boyera-Moore'a, 129
    wyszukiwania naiwnego, 120
    zachłannego, 102
  przeszukiwania wszerz, 86
  rekurencyjnego wyszukiwania binarnego, 41
  sortowania
    bąbelkowego, 38
    przez scalanie, 47, 50
    przez wybieranie, 39
    szybkiego, 45
  zasady niezgodności, 124

K

klasy złożoności problemów, 161
kod Huffmana, 99, 101
kody prefiksowe, 100
kolejka, 57
  dodawanie elementów, 58
  modelowanie, 60
  usuwanie elementów, 58
kolizja, 68
kompresja danych, 99

L

liczby pierwsze, 158
LIFO, Last In First Out, 58
lista powiązana, 51
  dwukierunkowa, 53
  operacje, 53
listy sąsiedztwa, 137

Ł

łańcuchowanie, 68

M

macierz sąsiedztwa, 139
mierzenie złożoności, 16
minimalne drzewa rozpinające, 159

N

najdłuższy wspólny podciąg, 114
najkrótsza ścieżka, 149, 154
notacja
  dużego O, 16, 22
  złożoności, 22

O

obliczanie najkrótszych ścieżek, 149
odległość między dwoma punktami, 106
odwracanie łańcucha znaków, 60
operacje
  na listach powiązanych, 53
  na strukturach danych, 51
optymalna podstruktura, 110

P

paradygmaty projektowania, 93
problem
  maksymalnego przepływu, 161
  maksymalnej podtablicy, 109
  najbliższej pary punktów, 106
  plecakowy, 111
  wydawania reszty, 116
problemy NP-zupełne, 162
programowanie dynamiczne, 110
projektowanie algorytmów, 93
przechodzenie przez graf, 142
przeszukiwanie
  w głąb, 144
  wszerz, 142
przypadek
  bazowy, 103
  rekurencyjny, 103

R

reguły notacji, 25
rekurencja, 40
  uniwersalna, 104
rekurencyjne wyszukiwanie binarne, 41
reprezentacja grafów, 136
rotacja drzewa, 89
rozkład na czynniki pierwsze, 158
rozwiązywanie kolizji, 71
  adresowanie otwarte, 71
  łańcuchowanie, 68

S

sekwencja próbkowania, 71
sito Eratostenesa, 158
słownik, 65
słowo kodowe, 99
sortowanie
  bąbelkowe, 35
    implementacja, 38
  przez scalanie, 46
    implementacja, 47, 50
  przez wybieranie, 39
  szybkie, 40
    implementacja, 45
stos, 58
  modelowanie, 60
struktura danych, 9, 50
  drzewo binarne, 78
  graf, 135
  kolejka, 57
  lista powiązana, 51
  słownik, 65
  stos, 58
  tablica, 60
    z haszowaniem, 65

T

tablica, 60
  z haszowaniem, 65
tworzenie
  kodu Huffmana, 100
  macierzy sąsiedztwa, 141

W

własność
  optymalnej podstruktury, 97
  zachłannego wyboru, 97
wspólne podproblemy, 111
wykładnik krytyczny, 105
wykrywanie cykli, 147
wyrażenie postfiksowe, 63
wyszukiwanie
  binarne, 41
  naiwne, 119, 120
  szybkie, 42
  wzorca, 119
wzorzec Boyera-Moore'a, 122

Z

zasada
  dobrego sufiksu, 125
  niezgodności, 123
złożoność
  algorytmu, 13, 16
  kwadratowa, 28
  liniowa, 27
  logarytmiczna, 29
  stała, 32
  wykładnicza, 30

Algorytm i struktura danych: tak działa optymalny kod!

Aby aplikacje mogły spełniać oczekiwania dotyczące wydajności i szybkości działania, programista musi orientować się w typowych problemach z wykonywaniem kodu i wiedzieć, które techniki sprawdzą się w danej sytuacji. W tym celu powinien biegle posługiwać się algorytmami i strukturami danych. Wiedza ta umożliwia rozpoznawanie typowych zagrożeń i wybór najlepszych rozwiązań. Warto pamiętać, że w przypadku większości codziennych problemów z kodem istnieją już wypróbowane rozwiązania. Znajomość tych zagadnień jest niezwykle ważna dla każdego inżyniera oprogramowania.

To książka przeznaczona dla programistów, którzy chcą w praktyczny sposób posługiwać się popularnymi algorytmami i strukturami danych, zrozumieć ich działanie i skuteczniej poprawiać wydajność swojego kodu w Javie. Przedstawiono tu narzędzia przydatne w pracy z algorytmami i w tworzeniu efektywnych aplikacji. Opisano praktyczne aspekty złożoności algorytmów. Omówiono algorytmy sortowania oraz inne popularne wzorce programowania, a także takie struktury danych jak drzewa binarne, tablice z haszowaniem i grafy. Następnie zaprezentowano koncepcje bardziej zaawansowane, wśród nich paradygmaty projektowania algorytmów i teorię grafów.

W tej książce między innymi:

- definiowanie algorytmu i złożoność algorytmiczna
- struktury danych i ich implementacje
- algorytmy sortowania i wyszukiwania wzorca w tekście
- paradygmaty projektowania algorytmów
- grafy i sposoby ich reprezentacji w programach komputerowych
- grafy jako moduły do budowy złożonych algorytmów

James Cutajar — jest programistą specjalizującym się w skalowalnych obliczeniach o wysokiej wydajności oraz w algorytmach rozproszonych. Pisze książki, bierze udział w projektach rozwoju otwartego oprogramowania, bloguje i zajmuje się marketingiem technologii.

ISBN 978-83-283-5329-9