Marcin Jamro

Struktury danych i algorytmy w języku C#

Projektowanie efektywnych aplikacji

Packt

Tytuł oryginału: C# Data Structures and Algorithms

Tłumaczenie: Krzysztof Bąbol

ISBN: 978-83-283-5047-2

Copyright © Packt Publishing 2018. First published in the English language under the title 'C# Data Structures and Algorithms (9781788833738)'

Polish edition copyright © 2019 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 Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/strdan
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 autorze 7

O recenzencie 8

Wstęp 9

Rozdział 1. Wprowadzenie 13

- **Język programowania** 14
- **Typy danych** 15
  - Typy wartościowe 16
  - Typy referencyjne 17
- **Instalacja i konfiguracja środowiska IDE** 22
- **Tworzenie projektu** 23
- **Wejście i wyjście** 25
  - Odczytywanie z wejścia 26
  - Zapisywanie do wyjścia 27
- **Uruchamianie i debugowanie** 30
- **Podsumowanie** 32

Rozdział 2. Tablice i listy 33

- **Tablice** 34
  - Tablice jednowymiarowe 34
  - Tablice wielowymiarowe 36
  - Tablice nieregularne 41
- **Algorytmy sortowania** 45
  - Sortowanie przez wybieranie 45
  - Sortowanie przez wstawianie 48
  - Sortowanie bąbelkowe 50
  - Sortowanie szybkie 52

Kup ksi k

Pole ksi k

Proste listy 55
Lista tablicowa 55
Lista generyczna 57
Przykład — średnia wartość 58
Przykład — lista osób 59
Listy uporządkowane 60
Przykład — książka adresowa 61
Listy wiązane 62
Przykład — czytnik książki 63
Listy cykliczne 66
Implementacja 67
Przykład — zakręć kołem 69
Podsumowanie 71

Rozdział 3. Stosy i kolejki 73

Stosy 73
Przykład — odwracanie wyrazów 75
Przykład — Wieże Hanoi 75
Kolejki 82
Przykład — telefoniczne biuro obsługi klienta z jednym konsultantem 84
Przykład — telefoniczne biuro obsługi klienta z wieloma konsultantami 88
Kolejki priorytetowe 92
Przykład — biuro telefonicznej obsługi klienta ze wsparciem priorytetowym 94
Podsumowanie 97

Rozdział 4. Słowniki i zbiory 99

Tablice z haszowaniem 99
Przykład — książka telefoniczna 101
Słowniki 104
Przykład — wyszukiwanie produktu 105
Przykład — dane użytkownika 107
Słowniki uporządkowane 109
Przykład — definicje 110
Zbiory haszowane 113
Przykład — kupony 115
Przykład — baseny 117
Zbiory „uporządkowane” 120
Przykład — usuwanie duplikatów 121
Podsumowanie 122

Rozdział 5. Warianty drzew 123

Zwykłe drzewa 124
Implementacja 124
Przykład — hierarchia identyfikatorów 126
Przykład — struktura przedsiębiorstwa 127
Drzewa binarne 129
Implementacja 132
Przykład — prosty quiz 136

Kup ksi k Pole ksi k

Binarne drzewa poszukiwań **139**
Implementacja 142
Przykład — wizualizacja drzewa BST 149
Drzewa AVL **156**
Implementacja 157
Przykład — utrzymuj zrównoważenie drzewa 158
Drzewa czerwono-czarne **159**
Implementacja 160
Przykład — funkcje drzew RBT 160
Kopce binarne **162**
Implementacja 163
Przykład — sortowanie przez kopcowanie 164
Kopce dwumianowe **165**
Kopce Fibonacciego **166**
Podsumowanie **168**

Rozdział 6. Odkrywanie grafów **169**

Koncepcja grafów **170**
Zastosowania **172**
Reprezentacja **173**
Lista sąsiedztwa 174
Macierz sąsiedztwa 175
Implementacja **178**
Węzeł 178
Krawędź 179
Graf 180
Przykład — krawędzie nieskierowane i nieważone 184
Przykład — krawędzie skierowane i ważone 185
Przeszukiwanie **186**
Przeszukiwanie w głąb 186
Przeszukiwanie wszerz 189
Minimalne drzewo rozpinające **192**
Algorytm Kruskala 193
Algorytm Prima 196
Przykład — kabel telekomunikacyjny 200
Kolorowanie **203**
Przykład — mapa województw 205
Najkrótsza ścieżka **207**
Przykład — mapa gry 210
Podsumowanie **213**

Kup ksi k Pole ksi k

Rozdział 7. Podsumowanie 215

Klasyfikacja struktur danych 215
Różnorodność zastosowań struktur danych 217
Tablice 217
Listy 218
Stosy 219
Kolejki 219
Słowniki 220
Zbiory 221
Drzewa 221
Kopce 222
Grafy 223
Słowo końcowe 224

Skorowidz 227

Kup książkę Poleć książkę

Tablice i listy

Jako programista z pewnością przechowywałeś w swoich aplikacjach różne kolekcje, takie jak dane użytkownika, książki albo logi. Jednym z naturalnych sposobów przechowywania takich danych jest użycie tablic i list. Czy jednak myślałeś kiedykolwiek o ich wariantach? Czy słyszałeś o tablicach nieregularnych albo o listach cyklicznych? W tym rozdziale zobaczysz działanie takich struktur danych wraz z przykładami i dokładnym opisem. To nie wszystko, ponieważ odniesiemy się do wielu zagadnień związanych z tablicami i listami, odpowiednich dla programistów o różnym poziomie umiejętności.

Na początku przedstawione zostaną tablice oraz ich podział na jednowymiarowe, wielowymiarowe i nieregularne. Poznasz cztery algorytmy sortowania, a mianowicie sortowanie przez selekcję, wstawianie, bąbelkowe oraz szybkie. W każdym przypadku zostanie zaprezentowany przykład opatrzony rysunkiem, kod implementacji oraz wyjaśnienie krok po kroku.

Tablice mają wiele zastosowań, ale jeszcze większe możliwości dają dostępne w języku C# listy generyczne. W pozostałej części rozdziału zobaczysz, jak korzystać z kilku wariantów list: prostych, uporządkowanych, dwukierunkowych i cyklicznych. Dla każdej odmiany zostanie przedstawiony kod w języku C# z dokładnym opisem.

Rozdział ten omawia:

- tablice,
- algorytmy sortowania,
- listy proste,
- listy uporządkowane,
- listy wiązane,
- listy cykliczne.

Kup ksi k

Pole ksi k

Tablice

Zacznijmy od tablicowej struktury danych. Używa się jej do przechowywania wielu zmiennych tego samego typu, jak `int`, `string` albo zdefiniowana przez użytkownika klasa. Jak wspomniałem we wstępie, podczas opracowywania aplikacji w języku C# można korzystać z kilku wariantów tablic, przedstawionych na poniższym rysunku. Dostępne są nie tylko tablice jednowymiarowe (oznaczone literą *a*), ale także wielowymiarowe (*b*) i nieregularne (*c*). Przykład każdej z nich jest pokazany poniżej:

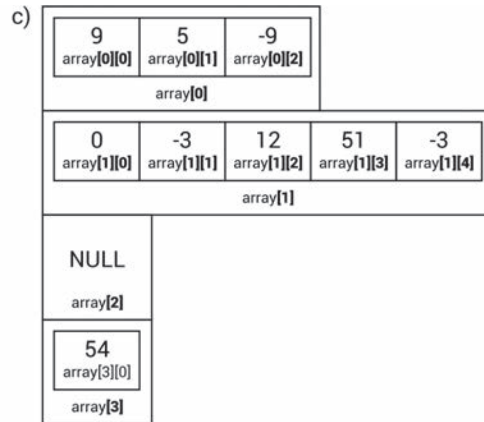
a)

9 array[0]
-11 array[1]
6 array[2]
-12 array[3]
1 array[4]

b)

9 array[0,0]	5 array[0,1]	-9 array[0,2]
-11 array[1,0]	4 array[1,1]	0 array[1,2]
6 array[2,0]	115 array[2,1]	3 array[2,2]
-12 array[3,0]	-9 array[3,1]	71 array[3,2]
1 array[4,0]	-6 array[4,1]	-1 array[4,2]

c)



Co istotne, po zainicjowaniu tablicy nie da się zmienić liczby jej elementów. Z tego powodu nie można w prosty sposób dodać na końcu tablicy nowego elementu ani wstawić go na określonej pozycji. Jeśli potrzebujesz takich funkcji, możesz użyć innych struktur danych opisanych w tym rozdziale — list generycznych.

> Więcej informacji o tablicach znajdziesz pod adresem *https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/arrays/*.

Po tym krótkim opisie jesteś gotowy, aby dowiedzieć się więcej o poszczególnych wariantach tablic i przyjrzeć się kodowi w języku C#. Przejdźmy więc do najprostszej odmiany tablic, czyli tablic jednowymiarowych.

Tablice jednowymiarowe

Tablica jednowymiarowa przechowuje kolekcję elementów tego samego typu, dostępnych za pomocą indeksu. Ważne, by pamiętać, że indeksy tablic w języku C# zaczynają się od zera, to znaczy pierwszy element ma indeks równy `0`, ostatni — rozmiar tablicy minus jeden.

Kup ksi k Pole ksi k

Przykładowa tablica jednowymiarowa jest pokazana na poprzednim rysunku (po jego lewej stronie, oznaczona literą *a*). Zawiera pięć elementów o wartościach 9, –11, 6, –12 i 1. Pierwszy element ma indeks równy 0, ostatni — 4.

Aby korzystać z tablicy jednowymiarowej, należy ją zadeklarować i zainicjować. Deklaracja jest bardzo prosta, bo wystarczy podać typ elementów i nazwę w następujący sposób:

```
typ[] nazwa;
```

Deklaracja tablicy o wartościach całkowitych wygląda następująco:

```
int[] numbers;
```

Wiesz już, jak zadeklarować tablicę, ale co z inicjowaniem? Aby zainicjować elementy tablicy domyślnymi wartościami, używa się operatora `new`, tak jak poniżej:

```
numbers = new int[5];
```

Deklarację i inicjowanie można oczywiście połączyć w jednej linii w następujący sposób:

```
int[] numbers = new int[5];
```

Niestety wszystkie elementy mają teraz domyślne wartości, to znaczy, w przypadku wartości całkowitych, zero. Należy więc nadać wartości poszczególnym elementom za pomocą operatora [] i indeksu elementu, co pokazuje listing:

```
numbers[0] = 9;
numbers[1] = -11; (...)
numbers[4] = 1;
```

Można ponadto połączyć deklarację i inicjowanie elementów tablicy określonymi wartościami przy użyciu jednego z poniższych wariantów kodu:

```
int[] numbers = new int[] { 9, -11, 6, -12, 1 };
int[] numbers = { 9, -11, 6, -12, 1 };
```

Jeśli elementy tablicy mają odpowiednie wartości, pobiera się je za pomocą operatora [] oraz indeksu, jak to zostało pokazane w poniższym wierszu:

```
int middle = numbers[2];
```

Wartość trzeciego elementu (o indeksie równym 2) tablicy o nazwie `numbers` jest tutaj pobierana i zapisywana w zmiennej `middle`.

> Więcej informacji o tablicach jednowymiarowych można uzyskać na stronie *https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/arrays/single-dimensional-arrays*.

Kup ksi k Pole ksi k

Przykład — nazwy miesięcy

Aby podsumować informacje o tablicach jednowymiarowych, rzućmy okiem na prosty przykład, w którym tablica służy do przechowywania nazw miesięcy. Nazwy tego typu powinny być pozyskiwane automatycznie, a nie wpisywane bezpośrednio do kodu źródłowego.

Oto implementacja:

```
string[] months = new string[12];

for (int month = 1; month <= 12; month++)
{
    DateTime firstDay = new DateTime(DateTime.Now.Year, month, 1);
    string name = firstDay.ToString("MMMM",
        CultureInfo.CreateSpecificCulture("pl"));
    months[month - 1] = name;
}
foreach (string month in months)
{
    Console.WriteLine($"-> {month}");
}
```

Na początku tworzy się tablicę jednowymiarową i inicjuje się ją domyślnymi wartościami. Zawiera ona `12` elementów przeznaczonych do przechowywania nazw miesięcy. Następnie pętla `for` przechodzi po numerach miesięcy od `1` do `12`. Tworzone są wystąpienia klasy `DateTime` reprezentujące pierwszy dzień każdego miesiąca.

Nazwa miesiąca jest pozyskiwana dzięki wywołaniu w wystąpieniu klasy `DateTime` metody `ToString` z odpowiednim formatem daty (`MMMM`) oraz określoną kulturą (w tym przypadku `pl`). Następnie nazwa miesiąca jest zachowywana w tablicy za pomocą operatora `[]` z indeksem elementu. Warto odnotować, że indeks jest równy bieżącej wartości zmiennej `month` minus jeden. Odjęcie jedynki jest niezbędne, bo pierwszy element tablicy ma wartość zero, a nie jeden.

Kolejnym interesującym fragmentem kodu jest pętla `foreach` przechodząca po wszystkich elementach tablicy. W przypadku każdego z nich w konsoli wyświetlana jest jedna linia, to znaczy nazwa miesiąca poprzedzona symbolem `->`. Wynik wygląda następująco:

```
-> January
-> February (...)
-> November
-> December
```

Jak wcześniej wspomniałem, tablice jednowymiarowe nie są jedynym dostępnym wariantem tablic. W kolejnym punkcie dowiesz się więcej o tablicach wielowymiarowych.

Tablice wielowymiarowe

Tablice w języku C# nie muszą ograniczać się tylko do jednego wymiaru. Istnieje również możliwość tworzenia tablic o dwóch, a nawet trzech wymiarach. Rozpocznijmy od przyjrzenia się deklaracji i inicjowaniu tablicy dwuwymiarowej o `5` rzędach i `2` kolumnach:

```
int[,] numbers = new int[5, 2];
```

Aby utworzyć tablicę trójwymiarową, używa się następującego kodu:

```
int[, ,] numbers = new int[5, 4, 3];
```

Można oczywiście połączyć deklarację z inicjowaniem, tak jak w poniższym przykładzie:

```
int[,] numbers = new int[,] =
{
    { 9, 5, -9 },
    { -11, 4, 0 },
    { 6, 115, 3 },
    { -12, -9, 71 },
    { 1, -6, -1 }
};
```

Przydałoby się drobne wyjaśnienie, w jaki sposób uzyskać dostęp do poszczególnych elementów tablicy dwuwymiarowej. Spójrzmy na poniższy przykład:

```
int number = numbers[2][1];
numbers[1][0] = 11;
```

W pierwszym wierszu kodu odczytywana jest wartość z trzeciego wiersza (indeks równy 2) i drugiej kolumny (indeks 1) tabeli (wartość ta wynosi 115), która jest przypisywana zmiennej `number`. Drugi wiersz kodu zmienia wartość w drugim wierszu i pierwszej kolumnie tabeli z -11 na 11.

> Więcej informacji o tablicach wielowymiarowych można uzyskać na stronie *https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/arrays/multidimensional-arrays*.

Przykład — tabliczka mnożenia

Pierwszy przykład ilustruje podstawowe operacje na tablicy dwuwymiarowej, prezentując tabliczkę mnożenia. Zapisane zostały wyniki mnożenia wszystkich liczb całkowitych od 1 do 10, co pokazują poniższe dane wyjściowe:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Przyjrzyjmy się sposobowi deklaracji i inicjowania tablicy:

```
int[,] results = new int[10, 10];
```

Tworzona jest tutaj tablica dwuwymiarowa o 10 wierszach i 10 kolumnach, a jej elementy są inicjowane wartościami domyślnymi, to znaczy zerami.

Kup ksi k Pole ksi k

Tablica jest gotowa, należy więc wypełnić ją wynikami mnożenia. Można tego dokonać za pomocą dwóch pętli `for`:

```
for (int i = 0; i < results.GetLength(0); i++)
{
    for (int j = 0; j < results.GetLength(1); j++)
    {
        results[i, j] = (i + 1) * (j + 1);
    }
}
```

Na powyższym listingu na obiekcie tablicy wywołuje się metodę `GetLength`. Metoda ta zwraca liczbę elementów konkretnego wymiaru, to znaczy pierwszego (gdy parametrem jest `0`) i drugiego (gdy parametrem jest `1`). W obu przypadkach zwracana jest wartość `10`, odpowiadająca wartościom podanym w czasie inicjowania tablicy.

Inną ważną kwestią jest sposób nadawania wartości elementom tablicy dwuwymiarowej. Aby tego dokonać, należy podać dwa indeksy: `results[i, j]`.

Na koniec wystarczy przedstawić wyniki. Można to zrobić za pomocą dwóch pętli `for`, tak jak w przypadku wypełniania tablicy. Ten fragment kodu przedstawia się następująco:

```
for (int i = 0; i < results.GetLength(0); i++)
{
    for (int j = 0; j < results.GetLength(1); j++)
    {
        Console.Write("{0,4}", results[i, j]);
    }
    Console.WriteLine();
}
```

Wyniki mnożenia po zamianie na wartości typu `string` mają różną długość, od jednego znaku (w przypadku `4` jako wyniku działania `2 * 2`) do trzech (`100` z mnożenia `10 * 10`). Dla lepszej prezentacji wyniku należy go zawsze wypisywać za pomocą `4` znaków. Jeśli zatem wartość całkowita zajmuje mniej miejsca, trzeba dodać na początku spacje. Na przykład wynik `1` będzie poprzedzony trzema spacjami (___1, gdzie _ oznacza spację), a `100` tylko jedną (_100). Aby osiągnąć ten cel, przy wywołaniu metody `Write` z klasy `Console` można użyć odpowiedniego złożonego ciągu formatującego (to znaczy `{0,4}`).

Przykład — mapa gry

Innym przykładem zastosowania tablicy dwuwymiarowej jest program przedstawiający mapę gry. Mapa jest prostokątem o 11 wierszach i 10 kolumnach. Każdy element tablicy określa typ terenu, taki jak trawa, piasek woda albo mur. Każde miejsce mapy powinno być wyświetlone w określonym kolorze (na przykład zielonym w przypadku trawy) za pomocą znaku obrazującego typ terenu (na przykład ≈ w przypadku wody), co pokazuje rysunek:

Kup ksi k

Pole ksi k

Na początku zadeklarujmy wartość wyliczenia o nazwie `TerrainEnum` z czterema stałymi, mianowicie `GRASS`, `SAND`, `WATER` i `WALL` w następujący sposób:

```
public enum TerrainEnum
{
    GRASS,
    SAND,
    WATER,
    WALL
}
```

> Dla lepszej czytelności całego projektu zaleca się zadeklarowanie typu `TerrainEnum` w oddzielnym pliku o nazwie *TerrainEnum.cs*. Należy stosować tę zasadę do wszystkich typów zdefiniowanych przez użytkownika, włącznie z klasami.

Następnie tworzone są dwie metody rozszerzeń, umożliwiające pobranie odpowiedniego koloru i znaku w zależności od typu terenu (odpowiednio `GetColor` i `GetChar`). Te metody rozszerzeń są zadeklarowane w klasie `TerrainEnumExtensions` w sposób przedstawiony poniżej:

```
public static class TerrainEnumExtensions
{
    public static ConsoleColor GetColor(this TerrainEnum terrain)
    {
        switch (terrain)
        {
            case TerrainEnum.GRASS: return ConsoleColor.Green;
            case TerrainEnum.SAND: return ConsoleColor.Yellow;
            case TerrainEnum.WATER: return ConsoleColor.Blue;
            default: return ConsoleColor.DarkGray;
        }
    }

    public static char GetChar(this TerrainEnum terrain)
    {
        switch (terrain)
        {
            case TerrainEnum.GRASS: return '\u201c';
            case TerrainEnum.SAND: return '\u25cb';
            case TerrainEnum.WATER: return '\u2248';
```


Kup ksi k Pole ksi k

```
                default: return '\u25cf';
            }
        }
    }
```

Warto wspomnieć, że metoda `GetChar` zwraca właściwy znak Unicode w oparciu o wartość `TerrainEnum`. Na przykład w przypadku stałej `WATER` zwracana jest wartość `'\u2248'` będąca reprezentacją znaku ≈.

> Czy słyszałeś o **metodach rozszerzeń** (ang. *extension methods*)? Jeśli nie, pomyśl o nich jako o metodach „dodanych” do jakiegoś istniejącego typu (zarówno wbudowanego, jak i zdefiniowanego przez użytkownika), które wywołuje się tak, jakby były zdefiniowane bezpośrednio w jego wystąpieniu. Deklarując metodę rozszerzenia, należy zdefiniować ją w klasie statycznej jako metodę statyczną z pierwszym parametrem wskazującym typ, do którego ma być „dodana”, poprzedzony słowem kluczowym `this`. Więcej informacji można znaleźć na stronie *https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/classes-and-structs/extension-methods*.

Przyjrzyjmy się treści metody `Main` w klasie `Program`. Konfiguruje ona mapę i prezentuje ją w konsoli za pomocą następującego kodu:

```
TerrainEnum[,] map =
{
    { TerrainEnum.SAND, TerrainEnum.SAND, TerrainEnum.SAND,
      TerrainEnum.SAND, TerrainEnum.GRASS, TerrainEnum.GRASS,
      TerrainEnum.GRASS, TerrainEnum.GRASS, TerrainEnum.GRASS,
      TerrainEnum.GRASS }, (...)
    { TerrainEnum.WATER, TerrainEnum.WATER, TerrainEnum.WATER,
      TerrainEnum.WATER, TerrainEnum.WATER, TerrainEnum.WATER,
      TerrainEnum.WATER, TerrainEnum.WALL, TerrainEnum.WATER,
      TerrainEnum.WATER }
};
Console.OutputEncoding = UTF8Encoding.UTF8;
for (int row = 0; row < map.GetLength(0); row++) {
    for (int column = 0; column < map.GetLength(1); column++)
    {
        Console.ForegroundColor = map[row, column].GetColor();
        Console.Write(map[row, column].GetChar() + " ");
    }
    Console.WriteLine();
}
Console.ForegroundColor = ConsoleColor.Gray;
```

Przyda się parę słów komentarza na temat sposobu pobierania koloru i pozyskiwania znaku dla konkretnego miejsca mapy. Obie operacje są wykonywane przez metody rozszerzeń dodane do zdefiniowanego przez użytkownika typu `TerrainEnum`. Z tego powodu najpierw pozyskiwana jest wartość wyliczenia `TerrainEnum` dla konkretnego miejsca mapy (za pomocą operatora [] i dwóch indeksów), a potem wywoływana odpowiednia metoda rozszerzenia, `GetChar` albo `GetColor`.

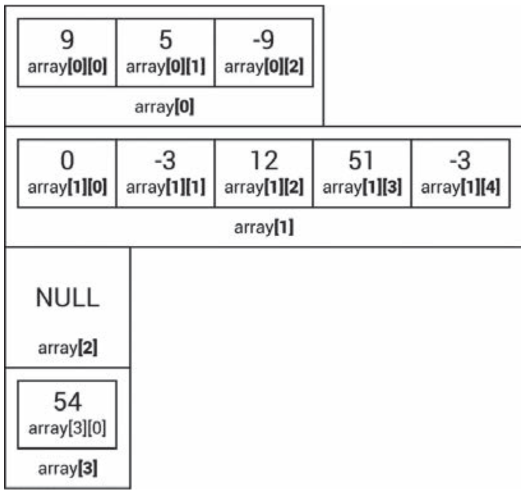
Do tej pory poznałeś zarówno tablice jedno-, jak i wielowymiarowe, ale do przedstawienia w książce pozostał jeszcze jeden wariant. Kontynuuj lekturę, aby dowiedzieć się o nim więcej.

Kup ksi k Pole ksi k

Tablice nieregularne

Ostatnim wariantem tablic opisanym w tej książce jest tablica nieregularna, zwana także **tablicą tablic**. Brzmi to skomplikowanie, ale na szczęście jest bardzo proste. Tablicę nieregularną można wyobrazić sobie jako tablicę jednowymiarową, której każdy element jest kolejną tablicą. Te wewnętrzne tablice mogą mieć oczywiście różną długość, a nawet mogą nie być zainicjowane.

Spójrz na poniższy rysunek, na którym zobaczysz przykładową tablicę nieregularną o czterech elementach. Pierwszy z nich jest tablicą z trzema elementami (9, 5, -9), drugi to tablica pięcioelementowa (0, -3, 12, 51, -3), trzeci nie jest zainicjowany (ma wartość NULL), a ostatni jest tablicą mającą tylko jeden element (54):



Zanim przejdziemy do przykładu, warto jeszcze wspomnieć o sposobie deklarowania i inicjowania tablicy nieregularnej, ponieważ jest on nieco inny niż w opisanych wcześniej tablicach. Spójrz na poniższy listing:

```
int[][] numbers = new int[4][];
numbers[0] = new int[] { 9, 5, -9 };
numbers[1] = new int[] { 0, -3, 12, 51, -3 };
numbers[3] = new int[] { 54 };
```

W pierwszym wierszu widać deklarację tablicy jednowymiarowej o czterech elementach. Każdy element jest następną tablicą jednowymiarową wartości całkowitych. Po wykonaniu pierwszego wiersza kodu tablica numbers jest inicjowana wartościami domyślnymi, to znaczy NULL. Z tego powodu należy ręcznie zainicjować poszczególne elementy, co widać w kolejnych trzech wierszach kodu. Warto zauważyć, że trzeci element nie jest zainicjowany.

Powyższy kod można też zapisać inaczej:

```
int[][] numbers =
{
    new int[] { 9, 5, -9 },
```


Kup ksi k Pole ksi k

```
        new int[] { 0, -3, 12, 51, -3 },
        NULL,
        new int[] { 54 }
    };
```

Wypada również pokrótce skomentować sposób dostępu do konkretnego elementu tablicy nieregularnej. Wygląda on następująco:

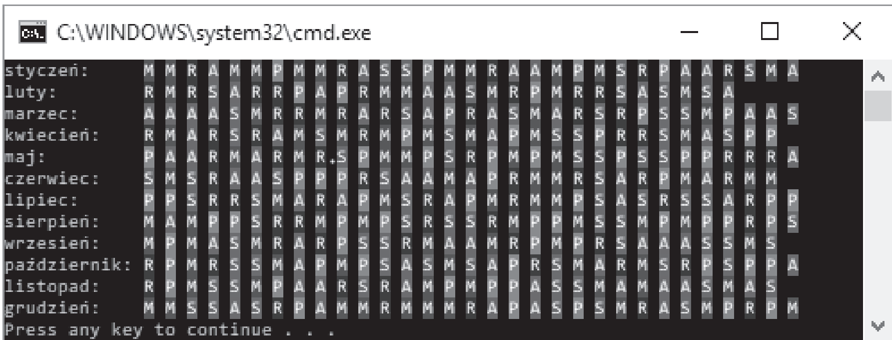
```
int number = numbers[1][2];
number[1][3] = 50;
```

Pierwszy wiersz kodu nadaje zmiennej `number` wartość `12`, czyli wartość trzeciego elementu (o indeksie `2`) tablicy, która jest drugim elementem tablicy nieregularnej. Drugi wiersz zmienia wartość czwartego elementu tablicy będącej drugim elementem tablicy nieregularnej z `51` na `50`.

> Więcej informacji o tablicach nieregularnych można uzyskać na stronie *https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/arrays/jagged-arrays*.

Przykład — roczny plan transportu

Po wprowadzeniu do tablic nieregularnych przejdźmy do przykładu. Zobaczysz, jak opracować program tworzący plan transportu na cały rok. Dla każdego dnia każdego miesiąca aplikacja rysuje jeden dostępny środek transportu. Na koniec program przedstawia wygenerowany plan, taki jak na poniższym rysunku:



Na początek zadeklarujmy typ wyliczeniowy ze stałymi reprezentującymi dostępne typy transportu, a mianowicie samochód, autobus, metro, rower i pieszą przechadzkę:

```
public enum TransportEnum
{
    CAR,
    BUS,
    SUBWAY,
    BIKE,
    WALK
}
```


Kup ksi k

Pole ksi k

W kolejnym kroku są tworzone dwie metody rozszerzeń, które zwracają znak i kolor reprezentujące dany środek transportu w konsoli. Oto ich kod:

```
public static class TransportEnumExtensions
{
    public static char GetChar(this TransportEnum transport)
    {
        switch (transport)
        {
            case TransportEnum.BIKE: return 'R';
            case TransportEnum.BUS: return 'A';
            case TransportEnum.CAR: return 'S';
            case TransportEnum.SUBWAY: return 'M';
            case TransportEnum.WALK: return 'P';
            default: throw new Exception("Nieznany środek transportu");
        }
    }

    public static ConsoleColor GetColor(
        this TransportEnum transport)
    {
        switch (transport)
        {
            case TransportEnum.BIKE: return ConsoleColor.Blue;
            case TransportEnum.BUS: return ConsoleColor.DarkGreen;
            case TransportEnum.CAR: return ConsoleColor.Red;
            case TransportEnum.SUBWAY:
                return ConsoleColor.DarkMagenta;
            case TransportEnum.WALK:
                return ConsoleColor.DarkYellow;
            default: throw new Exception("Nieznany środek transportu");
        }
    }
}
```

Powyższy kod nie wymaga dodatkowych wyjaśnień, ponieważ jest bardzo podobny do przedstawionego wcześniej. Przejdźmy teraz do metody `Main` klasy `Program`, która zostanie pokazana i opisana we fragmentach.

W pierwszej części tworzy się tablicę nieregularną i wypełnia się ją odpowiednimi wartościami. Zakładamy, że tablica nieregularna ma 12 elementów reprezentujących miesiące bieżącego roku. Każdy element jest jednowymiarową tablicą wartości typu `TransportEnum`. Długość takiej wewnętrznej tablicy zależy od liczby dni w danym miesiącu, wynosi na przykład 31 elementów dla stycznia i 30 elementów dla kwietnia. Oto pierwsza część kodu:

```
Random random = new Random(); int transportTypesCount =
    Enum.GetNames(typeof(TransportEnum)).Length;
TransportEnum[][] transport = new TransportEnum[12][];
for (int month = 1; month <= 12; month++)
{
    int daysCount = DateTime.DaysInMonth(
    DateTime.Now.Year, month);
    transport[month - 1] = new TransportEnum[daysCount];
```

```
        for (int day = 1; day <= daysCount; day++)
        {
            int randomType = random.Next(transportTypesCount);
            transport[month - 1][day - 1] = (TransportEnum)randomType;
        }
    }
```

Przeanalizujmy powyższy kod. Na początku jest tworzone nowe wystąpienie klasy `Random`, które posłuży potem do wylosowania jednego z dostępnych środków transportu. W następnym kroku z typu wyliczeniowego `TransportEnum` pobiera się stałą numeryczną będącą liczbą dostępnych typów transportu. W dalszej kolejności jest tworzona tablica nieregularna, a pętla `for` przechodzi po wszystkich miesiącach roku. W każdej iteracji pozyskuje się liczbę dni (za pomocą metody statycznej `DaysInMonth` klasy `DateTime`), a tablica (będąca elementem tablicy nieregularnej) jest inicjowana zerami. W kolejnym wierszu widać następną pętlę `for` służącą do przechodzenia po wszystkich dniach miesiąca. Wewnątrz tej pętli odbywa się losowanie typu transportu i zapisywanie go w odpowiednim elemencie tablicy, która jest elementem tablicy nieregularnej.

Następny fragment kodu jest związany z pokazaniem planu w konsoli:

```
string[] monthNames = GetMonthNames();
int monthNamesPart = monthNames.Max(n => n.Length) + 2;
for (int month = 1; month <= transport.Length; month++)
{
    Console.Write(
        $"{monthNames[month - 1]}:".PadRight(monthNamesPart));
    for (int day = 1; day <= transport[month - 1].Length; day++)
    {
        Console.ForegroundColor = ConsoleColor.White;
        Console.BackgroundColor =
            transport[month - 1][day - 1].GetColor();
        Console.Write(transport[month - 1][day - 1].GetChar());
        Console.BackgroundColor = ConsoleColor.Black;
        Console.ForegroundColor = ConsoleColor.Gray;
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

Na początku za pomocą metody `GetMonthNames`, która będzie opisana później, tworzona jest jednowymiarowa tablica z nazwami miesięcy. Następnie zmiennej `monthNamesPart` przypisuje się maksymalną długość tekstu potrzebnego do przechowania nazwy miesiąca. Aby odnaleźć tę długość, w kolekcji nazw miesięcy korzysta się z wyrażenia LINQ. Pozyskany wynik jest zwiększany o 2, by zarezerwować miejsce na przecinek i spację.

> Jedną ze świetnych funkcji języka C# jest mechanizm LINQ. Umożliwia on pobieranie danych w spójny sposób nie tylko z rozmaitych kolekcji, ale również z baz danych korzystających z języka **Structured Query Language** (**SQL**) i dokumentów w języku **Extensible Markup Language** (**XML**).
>
> Więcej informacji można uzyskać na stronie *https://docs.microsoft.com/pl-pl/dotnet/csharp/linq/index*.

Kup książkę Poleć książkę

Następnie używa się pętli for do przejścia po wszystkich elementach tablicy nieregularnej, czyli miesiącach. W każdej iteracji w konsoli pokazywana jest nazwa miesiąca.

Potem kolejna pętla for przechodzi po wszystkich elementach bieżącego elementu tablicy nieregularnej, czyli dniach miesiąca. Dla każdego z nich ustawiane są właściwe kolory (znaku i tła) oraz pokazywany jest odpowiedni znak.

Na koniec przyjrzyjmy się implementacji metody GetMonthNames:

```
private static string[] GetMonthNames()
{
    string[] names = new string[12];
    for (int month = 1; month <= 12; month++)
    {
        DateTime firstDay = new DateTime(
        DateTime.Now.Year, month, 1);
        string name = firstDay.ToString("MMMM",
        CultureInfo.CreateSpecificCulture("pl"));
        names[month - 1] = name;
    }
    return names;
}
```

Ten kod nie wymaga dodatkowych wyjaśnień, ponieważ bazuje na przykładzie dla tablic jednowymiarowych.

Algorytmy sortowania

Istnieje wiele algorytmów dokonujących rozmaitych operacji na tablicach, ale jednym z najczęstszych zadań jest sortowanie elementów tablicy w porządku rosnącym lub malejącym. Zagadnienie algorytmów sortowania obejmuje wiele metod, w tym sortowanie przez wybieranie, przez wstawianie, bąbelkowe oraz szybkie, które zostaną szczegółowo wyjaśnione w tym podrozdziale.

Sortowanie przez wybieranie

Rozpocznijmy od **sortowania przez wybieranie**, będącego jednym z najprostszych algorytmów. Ten algorytm dzieli tablicę na dwie części, to znaczy posortowaną i nieposortowaną. Podczas kolejnych iteracji algorytm znajduje najmniejszy element części nieposortowanej i zamienia go miejscami z pierwszym elementem tej części. Brzmi to bardzo prosto, prawda?

Aby lepiej zrozumieć ten algorytm, spójrz na kolejne iteracje w przypadku tablicy dziewięcioelementowej (*–11*, *12*, *–42*, *0*, *1*, *90*, *68*, *6*, *–9*), które przedstawia poniższy rysunek:

Kup ksi k Pole ksi k

Dla ułatwienia analizy grubą linią została zaznaczona granica pomiędzy posortowaną i nieposortowaną częścią tablicy. Na początku (*krok 1.*) granica jest zlokalizowana na samym szczycie tablicy, co oznacza, że posortowana część jest pusta. Algorytm znajduje zatem najmniejszą wartość w części nieposortowanej (*–42*) i zamienia ją z pierwszym elementem tej części (*–11*). Wynik został przedstawiony w *kroku 2.*, w którym część posortowana zawiera jeden element (*–42*), a część nieposortowana składa się z ośmiu elementów. Wspomniane kroki są wykonywane dotąd, aż w części nieposortowanej zostanie tyko jeden element. Ostateczny wynik został zaprezentowany w *kroku 9*.

Poznałeś działanie algorytmu sortowania przez wybieranie, ale czy wiesz, jaka jest rola wskaźników `i` oraz `m` pokazanych po lewej stronie kolejnych kroków na powyższym rysunku? Odnoszą się one do zmiennych użytych w implementacji tego algorytmu. Nadszedł zatem czas, aby zobaczyć kod w języku C#.

Algorytm został zaimplementowany w postaci statycznej klasy `SelectionSort` z generyczną metodą statyczną `Sort` pokazaną na poniższym listingu:

```
public static class SelectionSort {
    public static void Sort<T>(T[] array) where T : IComparable
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            int minIndex = i;
            T minValue = array[i];
            for (int j = i + 1; j < array.Length; j++)
            {
                if (array[j].CompareTo(minValue) < 0)
                {
                    minIndex = j;
                    minValue = array[j];
                }
            }
            Swap(array, i, minIndex);
        }
    } (...)
}
```


Kup ksi k
Pole ksi k

Metoda `Sort` przyjmuje jeden parametr, mianowicie tablicę do posortowania (`array`). Wewnątrz tej metody pętla `for` przechodzi po elementach tablicy do momentu, gdy w części nieposortowanej zostanie tylko jeden element. Tak więc liczba powtórzeń pętli jest równa długości tablicy minus jeden (`array.Length - 1`). W ramach każdej iteracji używa się kolejnej pętli `for` do znalezienia najmniejszej wartości w części nieposortowanej (`minValue`, od indeksu `i + 1` aż do końca tablicy) oraz do zachowania indeksu najmniejszej wartości (`minIndex`, na powyższym rysunku oznaczony jako wskaźnik `m`). Następnie zamienia się miejscami najmniejszy element części nieposortowanej (o indeksie równym `minIndex`) z pierwszym elementem tej części (o indeksie równym `i`) za pomocą dodatkowej metody `Swap`, której implementacja wygląda tak jak poniżej:

```
private static void Swap<T>(T[] array, int first, int second) {
    T temp = array[first];
    array[first] = array[second];
    array[second] = temp;
}
```

Jeśli chcesz przetestować implementację algorytmu sortowania przez wybieranie, umieść poniższy kod w metodzie `Main` klasy `Program`:

```
int[] integerValues = { -11, 12, -42, 0, 1, 90, 68, 6, -9 };
SelectionSort.Sort(integerValues);
Console.WriteLine(string.Join(" | ", integerValues));
```

Powyższy kod deklaruje i inicjuje nową tablicę. Następnie wywołuje statyczną metodę `Sort` z tablicą jako parametrem. Na koniec przez połączenie elementów tablicy (rozdzielonych znakiem `|`) tworzona i pokazywana w konsoli jest wartość typu `string`:

```
-42 | -11 | -9 | 0 | 1 | 6 | 12 | 68 | 90
```

Dzięki wykorzystaniu metody generycznej można z łatwością użyć tej klasy do sortowania różnych tablic, zawierających na przykład liczby zmiennoprzecinkowe albo ciągi. Oto przykładowy kod:

```
string[] stringValues = { "Maria", "Marcin", "Anna", "Jakub", "Jerzy", "Nikola" };
SelectionSort.Sort(stringValues);
Console.WriteLine(string.Join(" | ", stringValues));
```

Na wyjściu pojawią się poniższe dane:

```
Anna | Jakub | Jerzy | Marcin | Maria | Nikola
```

Przy omawianiu rozmaitych algorytmów jednym z najważniejszych zagadnień jest **złożoność obliczeniowa** (ang. *computational complexity*), zwłaszcza **złożoność czasowa** (ang. *time complexity*). Ma ona kilka wariantów, na przykład dla najgorszego albo średniego przypadku. Złożoność można interpretować jako liczbę podstawowych operacji, które musi wykonać algorytm w zależności od rozmiaru danych wejściowych (n). Wyraża się ją za pomocą **notacji „duże O”** (ang. *Big O notation*), na przykład jako $O(n)$, $O(n^2)$ albo $O(n \log(n))$. Co to znaczy? Notacja $O(n)$ wskazuje, że liczba operacji wzrasta liniowo wraz z rozmiarem danych wejściowych (n). Rząd złożoności $O(n^2)$ jest nazywany **kwadratowym** (ang. *quadratic*), a $O(n \log(n))$ **liniowo-logarytmicznym** (ang. *linearithmic*). Istnieją także inne rzędy złożoności, na przykład $O(1)$, czyli **stały**.

Kup książkę Poleć książkę

W przypadku sortowania przez wybieranie zarówno pesymistyczna, jak i średnia złożoność czasowa wynosi $O(n^2)$. Dlaczego? Aby odpowiedzieć na to pytanie, przyjrzyjmy się kodowi. Są tam dwie pętle (jedna wewnątrz drugiej), a każda z nich przechodzi przez wiele elementów tablicy. Z tego powodu złożoność jest wykazywana jako $O(n^2)$.

> Więcej informacji na temat sortowania przez wybieranie i jego implementacji można znaleźć na stronach:
>
> - *https://pl.wikipedia.org/wiki/Sortowanie_przez_wybieranie*.
> - *http://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Selection_sort*.

Właśnie nauczyłeś się pierwszego algorytmu sortowania! Jeśli jesteś zainteresowany poznaniem kolejnej metody, przejdź do następnego punktu, przedstawiającego sortowanie przez wstawianie.

Sortowanie przez wstawianie

Sortowanie przez wstawianie (ang. *insertion sort*) jest kolejnym algorytmem pozwalającym w prosty sposób uporządkować tablicę jednowymiarową, co widać na poniższym rysunku. Podobnie jak w przypadku sortowania przez wybieranie, tablica jest dzielona na dwie części, to znaczy posortowaną i nieposortowaną, na początku pierwszy element zalicza się jednak do części posortowanej. W każdej iteracji algorytm bierze pierwszy element z części nieposortowanej i umieszcza go w odpowiednim miejscu części posortowanej, zostawiając ją we właściwym porządku. Te operacje są powtarzane tak długo, aż część nieposortowana będzie pusta.

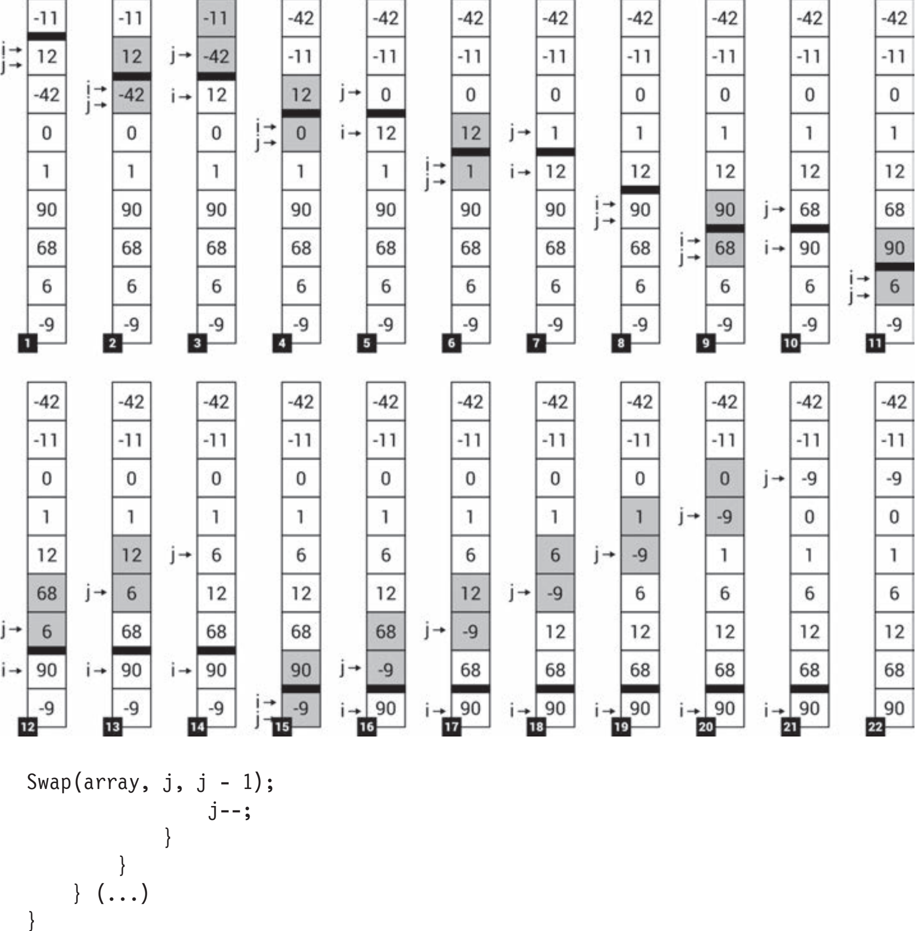
Przyjrzyjmy się przykładowi sortowania tablicy dziewięcioelementowej (*–11*, *12*, *–42*, *0*, *1*, *90*, *68*, *6*, *–9*) za pomocą sortowania przez wstawianie, przedstawionemu na rysunku na następnej stronie.

Na początku w części posortowanej znajduje się tylko jeden element (*–11 — krok 1.*). Potem jest znajdowany najmniejszy element części nieposortowanej (*–42*) i w serii zamian jest on przenoszony na właściwe miejsce w części posortowanej, czyli na początek tablicy (*kroki 2.* i *3.*). W ten sposób część posortowana zwiększyła się do dwóch elementów, czyli *–42* i *–11*. Te operacje są powtarzane dotąd, aż część nieposortowana będzie pusta (*krok 22.*).

Kod implementujący sortowanie przez wstawianie jest bardzo prosty:

```
public static class InsertionSort {
    public static void Sort<T>(T[] array) where T : IComparable
    {
        for (int i = 1; i < array.Length; i++)
        {
            int j = i;
            while (j > 0 && array[j].CompareTo(array[j - 1]) < 0)
            {
```


Kup ksi k Pole ksi k




```
Swap(array, j, j - 1);
                j--;
            }
        }
    } (...)
}
```

Podobnie jak w przypadku sortowania przez wybieranie, implementację zawarto w nowej klasie o nazwie `InsertionSort`. Generyczna metoda statyczna `Sort` dokonuje operacji sortowania, przyjmując jako parametr tablicę. Wewnątrz tej metody pętla `for` przechodzi przez wszystkie elementy w części nieposortowanej. Na początku zmienna `i` przyjmuje wartość nie `0`, ale `1`. W każdym powtórzeniu pętli `for` uruchamiana jest pętla `while` przenosząca pierwszy element w nieposortowanej części tablicy (o indeksie równym wartości zmiennej `i`) na właściwe miejsce w części posortowanej przy użyciu pomocniczej metody `Swap`, zaimplementowanej tak samo jak w przypadku sortowania przez wybieranie. Sposób testowania sortowania przez wybieranie jest również bardzo podobny, ale należy użyć innej nazwy klasy, czyli `InsertionSort` zamiast `SelectionSort`.

> Więcej informacji na temat sortowania przez wstawianie i jego implementacji można znaleźć na stronach:
>
> - *https://pl.wikipedia.org/wiki/Sortowanie_przez_wstawianie.*
> - *https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Insertion_sort.*

Kup ksi k

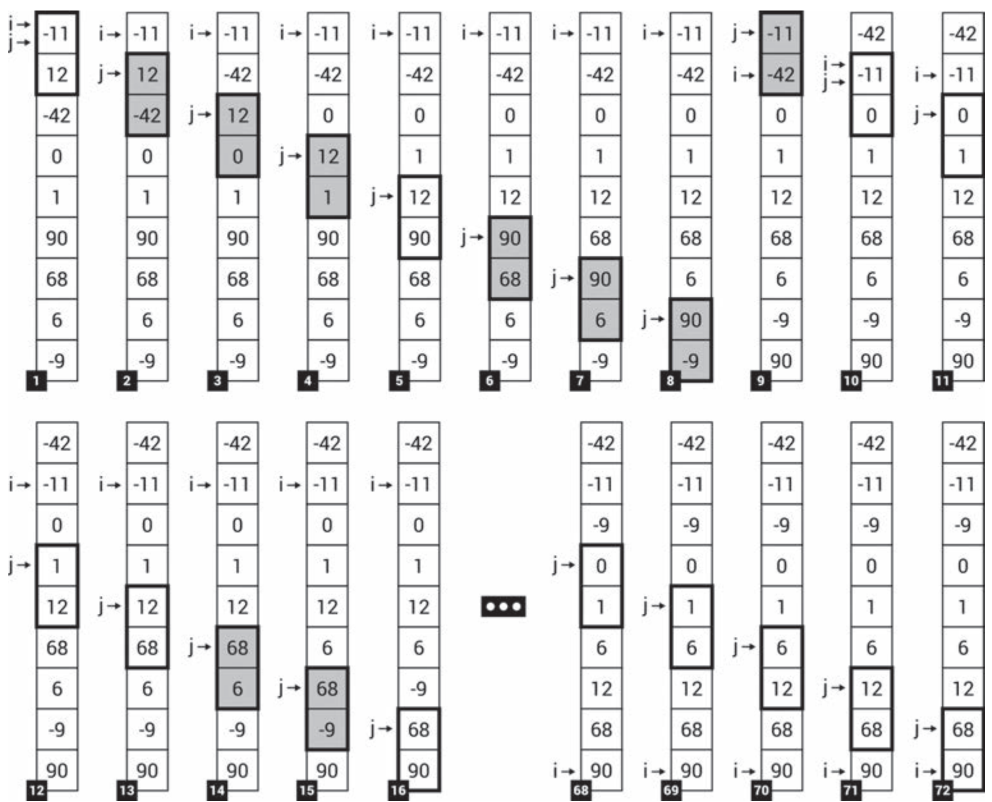
Pole ksi k

Na koniec warto wspomnieć o złożoności czasowej sortowania przez wstawianie. Podobnie jak w przypadku sortowania przez wybieranie, zarówno pesymistyczna, jak i średnia złożoność czasowa wynosi $O(n^2)$. Gdy przyjrzysz się kodowi, również zobaczysz dwie pętle (`for` i `while`), umieszczone jedna w drugiej, których wykonanie może się powtarzać wiele razy w zależności od rozmiaru danych wejściowych.

Sortowanie bąbelkowe

Trzecim algorytmem sortowania przedstawionym w książce będzie **sortowanie bąbelkowe** (ang. *bubble sort*). Jego sposób działania jest bardzo prosty, ponieważ algorytm zwyczajnie przechodzi przez tablicę i porównuje sąsiednie elementy. Jeśli są umieszczone w niewłaściwej kolejności, to są zamieniane miejscami. Brzmi to bardzo prosto, ale algorytm nie jest zbyt efektywny, a jego użycie z dużymi kolekcjami może spowodować problemy z wydajnością.

Aby lepiej zrozumieć, jak działa opisany algorytm, przyjrzyj się poniższemu rysunkowi przedstawiającemu operację sortowania jednowymiarowej tablicy o dziewięciu elementach (*–11*, *12*, *–42*, *0*, *1*, *90*, *68*, *6*, *–9*):



Kup książkę Poleć książkę

Jak widać, w każdym kroku algorytm porównuje dwa sąsiednie elementy tablicy i zamienia je miejscami, jeśli to niezbędne. Na przykład w *kroku 1.* porównywane są wartości *–11* i *12*, ale są one umieszczone we właściwym porządku, nie trzeba ich więc zamieniać. W *kroku 2.* porównywane są kolejne sąsiednie elementy (to znaczy *12* i *–42*). Tym razem te elementy nie są umieszczone we właściwym porządku, są więc zamieniane miejscami. Wspomniane operacje są wykonywane wiele razy. Na końcu, w *kroku 72.*, tablica będzie posortowana.

Algorytm wydaje się bardzo prosty, a jak wygląda jego implementacja? Czy jest również tak prosta? Na szczęście tak! Wystarczy użyć dwóch pętli, porównać sąsiednie elementy i w razie konieczności zamienić je miejscami. To wszystko! Przyjrzyj się poniższemu listingowi:

```
public static class BubbleSort {
    public static void Sort<T>(T[] array) where T : IComparable
    {
        for (int i = 0; i < array.Length; i++)
        {
            for (int j = 0; j < array.Length - 1; j++)
            {
                if (array[j].CompareTo(array[j + 1]) > 0)
                {
                    Swap(array, j, j + 1);
                }
            }
        }
    } (...)
}
```

Implementację algorytmu sortowania bąbelkowego zawiera generyczna metoda statyczna `Sort`, zadeklarowana w klasie `BubbleSort`. Jak już wspomniałem, użyto w niej dwóch pętli `for` z porównaniem i wywołaniem metody `Swap` (zaimplementowanej tak samo jak w przypadku wcześniej opisanych algorytmów sortowania). Poza tym do przetestowania implementacji można wykorzystać podobny kod, ale trzeba pamiętać o zmianie nazwy klasy na `BubbleSort`.

Algorytm sortowania bąbelkowego daje się zoptymalizować dzięki prostej modyfikacji. Opiera się ona na założeniu, że jeśli w ciągu jednego przejścia przez tablicę nie wykryto żadnych zmian, to porównania należy zatrzymać. Oto zmodyfikowany kod:

```
public static T[] Sort<T>(T[] array) where T : IComparable {
    for (int i = 0; i < array.Length; i++)
    {
        bool isAnyChange = false;
        for (int j = 0; j < array.Length - 1; j++)
        {
            if (array[j].CompareTo(array[j + 1]) > 0)
            {
                isAnyChange = true;
                Swap(array, j, j + 1);
            }
        }
        if (!isAnyChange)
        {
```


Kup ksi k

Pole ksi k

```
                break;
            }
        }
        return array;
    }
```

Dzięki tej prostej modyfikacji liczbę porównań można znacznie ograniczyć. W powyższym przykładzie zmniejszyła się ona z 72 do 56.

> Więcej informacji na temat sortowania bąbelkowego oraz jego implementacji można znaleźć na stronach:
>
> - *https://pl.wikipedia.org/wiki/Sortowanie_bąbelkowe*.
> - *https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Bubble_sort*.

Przed przejściem do omówienia następnego algorytmu warto wspomnieć o złożoności czasowej sortowania bąbelkowego. Jak już zapewne zgadłeś, zarówno w najgorszym, jak i średnim przypadku jest ona taka sama jak w sortowaniu przez wybieranie i wstawianie, czyli $O(n^2)$.

Sortowanie szybkie

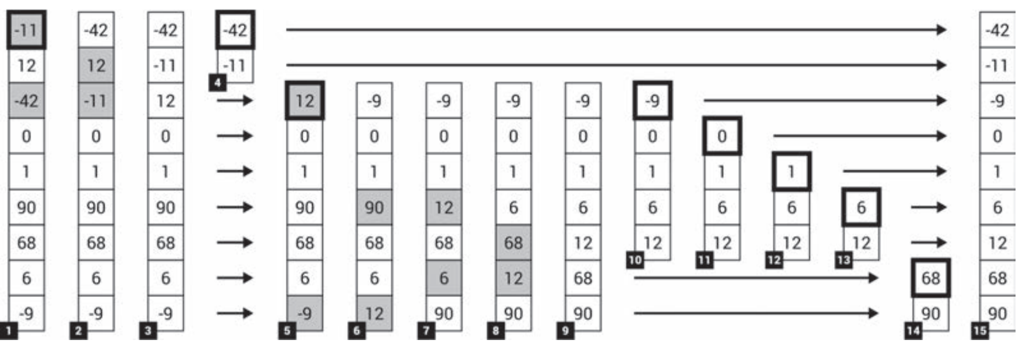
Ostatni algorytm sortowania opisany w tej książce nosi nazwę **sortowania szybkiego** (ang. *quicksort*). Jest to jeden z popularnych **algorytmów „dziel i rządź”** (ang. *divide and conquer*), który dzieli większy problem na zestaw mniejszych. W dodatku oferuje on programistom wydajny sposób sortowania. Czy to znaczy, że jego koncepcja i implementacja są bardzo skomplikowane? Na szczęście nie! W tym punkcie dowiesz się, jak działa ten algorytm i jak może wyglądać jego implementacja. Zaczynajmy!

Jak działa algorytm? Na początku wybiera pewną wartość (na przykład z pierwszego lub środkowego elementu tablicy) jako **element osiowy** (ang. *pivot*). Potem przestawia tablicę w taki sposób, aby wartości mniejsze lub równe elementowi osiowemu były umieszczone przed nim (tworząc początkową podtablicę), a wartości większe od elementu osiowego po nim (w końcowej podtablicy). Ten proces to **podział na partycje** (ang. *partitioning*). W tej książce używany jest **podział Hoare'a**. Następnie algorytm rekursywnie sortuje każdą z wcześniej wymienionych podtablic. Oczywiście każda podtablica jest w dalszym ciągu dzielona na następne dwie i tak dalej. Rekursywne wywołania kończą się, jeśli podtablica ma zero elementów, ponieważ wtedy nie ma nic do sortowania.

Powyższy opis może wydawać się nieco skomplikowany, spójrzmy więc na przykład (zobacz rysunek na następnej stronie).

Przykład ten pokazuje, jak algorytm sortowania szybkiego porządkuje jednowymiarową tablicę o dziewięciu elementach (*–11*, *12*, *–42*, *0*, *1*, *90*, *68*, *6*, *–9*). W tym przypadku jako wartość osiową przyjęto wartość pierwszego elementu aktualnie sortowanej podtablicy. W *kroku 1.* jako wartość osiową wybrano *–11*. Konieczne jest przestawienie tablicy po to, by w początkowej

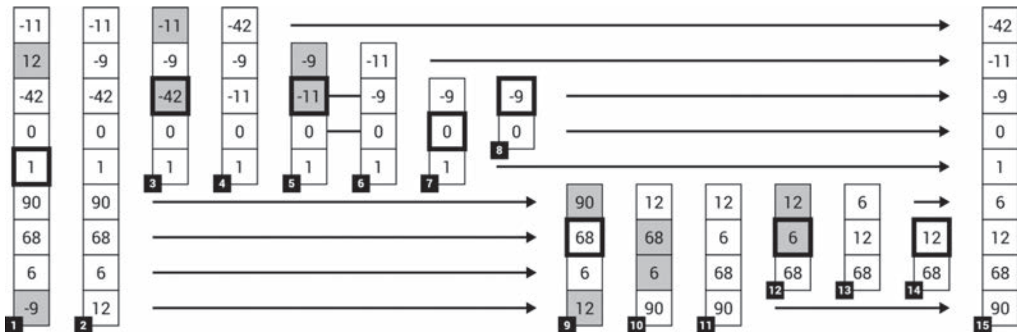
Kup ksi k Pole ksi k



podtablicy pozostały tylko wartości mniejsze albo równe elementowi rozdzielającemu (*–42*, *–11*), a w końcowej tylko wartości od niego większe (*12*, *0*, *1*, *90*, *68*, *6*, *–9*), dlatego *–11* jest zamieniane miejscami z *–42*, a *12* z *–11*. Potem algorytm jest wywoływany rekursywnie dla obu podtablic, to znaczy (*–42*, *11*) i (*12*, *0*, *1*, *90*, *68*, *6*, *–9*), te podtablice są więc poddawane tej samej analizie co tablica wejściowa.

Na przykład w *kroku 5.* jako wartość osiową wybrano *12*. Podtablica jest dzielona na dwie, to znaczy (*–9*, *0*, *1*, *6*, *12*) i (*68*, *90*). W obu podtablicach wybiera się elementy osiowe, czyli *–9* i *68*. Po wykonaniu takich operacji we wszystkich pozostałych fragmentach tablicy uzyskuje się wynik końcowy, pokazany z prawej strony rysunku (*krok 15.*).

Warto wspomnieć, że w innych implementacjach tego algorytmu rozmaicie dobiera się element osiowy. Spójrzmy na przykład, jak zmienią się kolejne kroki, jeśli wybierzemy wartość środkowego elementu:



Jeśli zrozumiałeś działanie algorytmu, zapoznaj się z implementacją. Jest ona bardziej skomplikowana niż w podanych wcześniej przykładach i przy wywoływaniu metody sortującej podtablice korzysta z **rekurencji** (ang. *recursion*). Kod został umieszczony w klasie `QuickSort`:

```
public static class QuickSort
{
    public static void Sort<T>(T[] array) where T : IComparable
    {
        Sort(array, 0, array.Length - 1);
    } (...)
}
```


Kup ksi k

Pole ksi k

Klasa QuickSort zawiera dwie odmiany metody Sort. Pierwsza przyjmuje tylko jeden parametr, to znaczy tablicę do posortowania, i jest pokazana na powyższym listingu. Jedyne, co robi, to wywołuje drugi wariant metody Sort, pozwalający określić indeks początkowy i końcowy fragmentu tablicy do posortowania. Oto druga wersja metody Sort:

```
private static T[] Sort<T>(T[] array, int lower, int upper)
    where T : IComparable
{
    if (lower < upper)
    {
        int p = Partition(array, lower, upper);
        Sort(array, lower, p);
        Sort(array, p + 1, upper);
    }
    return array;
}
```

Metoda Sort sprawdza, czy tablica (lub podtablica) ma co najmniej dwa elementy, porównując wartości zmiennych lower i upper. Jeśli tak, wywołuje ona metodę Partition, odpowiedzialną za podział na partycje, a potem rekursywnie wywołuje metodę Sort dla obu podtablic, to znaczy początkowej (o indeksach od lower do p) i końcowej (od p + 1 do upper).

Oto kod dzielący tablicę na partycje:

```
private static int Partition<T>(T[] array, int lower, int upper)
    where T : IComparable
{
    int i = lower;
    int j = upper;
    T pivot = array[lower];
    // albo: T pivot = array[(lower + upper) / 2];
    do
    {
        while (array[i].CompareTo(pivot) < 0) { i++; }
        while (array[j].CompareTo(pivot) > 0) { j--; }
        if (i >= j) { break; }
        Swap(array, i, j);
    }
    while (i <= j);
    return j;
}
```

Na początku wartość osiowa jest wybierana i zapisywana w zmiennej pivot. Jak już wiesz, można ją wybrać rozmaicie, na przykład biorąc wartość pierwszego elementu (co pokazuje powyższy listing), wartość środkowego elementu (co pokazuje komentarz w powyższym kodzie), a nawet wartość losową. Następnie w pętli do-while tablica jest przekształcana zgodnie ze schematem Hoare'a za pomocą porównań i zamiany elementów. Na koniec jest zwracana aktualna wartość zmiennej j.

Jak wygląda złożoność czasowa sortowania szybkiego? Czy sądzisz, że jest inna niż w sortowaniu przez wybieranie, wstawianie oraz bąbelkowym? Jeśli tak, masz rację! Sortowanie szybkie ma przeciętną złożoność czasową $O(n \log(n))$, choć w najgorszym przypadku wynosi ona $O(n^2)$.

> Zaprezentowana implementacja jest oparta na schemacie podziału na partycje Hoare'a, którego pseudokod i objaśnienie przedstawia strona *https://en.wikipedia.org/wiki/Quicksort*. Istnieją różne sposoby implementacji sortowania szybkiego. Więcej informacji można znaleźć na stronie *https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Quicksort*.

Proste listy

Tablice są naprawdę użytecznymi strukturami danych i wykorzystuje się je w wielu algorytmach. Niemniej jednak w niektórych przypadkach ich zastosowanie jest skomplikowane, ponieważ nie można zwiększać ani zmniejszać wielkości już utworzonej tablicy. Co zrobić, jeśli całkowita liczba elementów do przechowania w kolekcji jest nieznana? Czy utworzyć bardzo dużą tablicę i po prostu nie używać zbędnych elementów? To rozwiązanie nie wygląda dobrze, prawda? Znacznie lepszym podejściem jest wykorzystanie struktury danych pozwalającej w miarę potrzeb dynamicznie zwiększać rozmiar kolekcji.

Lista tablicowa

Pierwszą strukturą danych, która spełnia to wymaganie, jest **lista tablicowa** (ang. *array list*), reprezentowana przez klasę `ArrayList` z przestrzeni nazw `System.Collections`. Klasa ta służy do przechowywania dużych kolekcji danych, do których w razie potrzeby można z łatwością dodać nowy element. Oczywiście da się też usuwać elementy, liczyć je oraz znajdować indeks konkretnej wartości zapisanej w liście tablicowej.

W jaki sposób się to robi? Spójrzmy na poniższy kod:

```
ArrayList tablicaList = nowy ArrayList();
tablicaList.Add(5);
tablicaList.AddRange(new int[] { 6, -7, 8 });
tablicaList.AddRange(new object[] { "Marcin", "Maria" });
tablicaList.Insert(5, 7.8);
```

W pierwszym wierszu tworzone jest nowe wystąpienie klasy `ArrayList`. Następnie w celu dodania nowych elementów do listy tablicowej używa się metod `Add`, `AddRange` i `Insert`. Pierwsza z nich (czyli `Add`) pozwala dodać nowy element na końcu listy. Metoda `AddRange` dodaje na końcu listy tablicowej kolekcję elementów, `Insert` zaś umieszcza element w podanym miejscu. Po wykonaniu powyższego kodu lista tablicowa będzie zawierać następujące elementy: `5`, `6`, `-7`, `8`, `"Marcin"`, `7.8` i `"Maria"`. Jak widać, wszystkie przechowywane w niej elementy są typu `object`. Można zatem jednocześnie umieszczać w tej samej kolekcji dane różnych typów.

> Jeśli chcesz określić typ każdego elementu przechowywanego w liście, możesz skorzystać z generycznej klasy `List`, która jest opisana zaraz po `ArrayList`.

Kup ksi k Pole ksi k

Warto wspomnieć, że można łatwo uzyskać dostęp do konkretnego elementu listy tablicowej, używając po prostu indeksu, co pokazują poniższe dwa wiersze kodu:

```
object first = arrayList[0];
int third = (int)arrayList[2];
```

Przyjrzyjmy się rzutowaniu do typu `int` w drugim wierszu. Takie rzutowanie jest konieczne, ponieważ lista tablicowa przechowuje wartości typu `object`. W celu uzyskania dostępu do konkretnych elementów kolekcji tak samo jak w przypadku tablic używa się indeksów zaczynających się od zera.

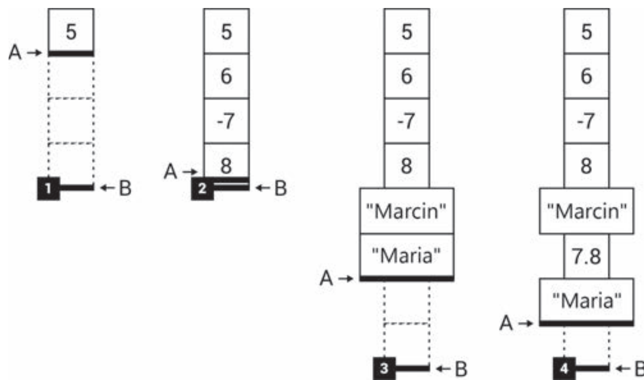
Do przejścia przez wszystkie elementy można oczywiście skorzystać z pętli `foreach`:

```
foreach (object element in arrayList) {
    Console.WriteLine(element);
}
```

To nie wszystko! Klasa `ArrayList` ma zestaw właściwości i metod, których można użyć, opracowując aplikacje korzystające z tej struktury danych. Na początek przyjrzyjmy się właściwościom `Count` i `Capacity`:

```
int count = arrayList.Count;
int capacity = arrayList.Capacity;
```

Pierwsza z nich (`Count`) zwraca liczbę elementów listy tablicowej, a druga (`Capacity`) wskazuje, ile elementów można w niej przechowywać. Jeśli sprawdzisz właściwość `Capacity` po dodaniu nowych elementów do listy tablicowej, zobaczysz, że jej wartość jest automatycznie zwiększana, aby przygotować miejsce na nowe elementy. Pokazuje to poniższy rysunek, obrazujący różnicę między właściwością `Count` (oznaczoną literą *A*) a `Capacity` (*B*):



Następnym często wykonywanym i ważnym zadaniem jest sprawdzenie, czy lista tablicowa zawiera element o określonej wartości. Aby wykonać tę operację, wywołuje się metodę `Contains`, co pokazuje poniższy wiersz kodu:

```
bool containsMary = arrayList.Contains("Maria");
```


Kup ksi k Pole ksi k

Jeśli w liście tablicowej znajduje się podana wartość, zwracana jest wartość `true`, w przeciwnym przypadku `false`. Za pomocą tej metody można sprawdzić, czy element istnieje w kolekcji. Jak jednak odnaleźć indeks elementu? Przy użyciu metody `IndexOf` albo `LastIndexOf`, tak jak w poniższym wierszu kodu:

```
int minusIndex = arrayList.IndexOf(-7);
```

Metoda `IndexOf` zwraca indeks pierwszego wystąpienia elementu w liście tablicowej, a `LastIndexOf` — indeks ostatniego. Jeśli nie znaleziono wartości, metoda zwraca `-1`.

Oprócz dodawania elementów do listy tablicowej można również z łatwością usuwać z niej dodane elementy, co pokazuje poniższy kod:

```
arrayList.Remove(5);
```

Do usunięcia elementów z listy tablicowej można użyć więcej niż jednej metody, a mianowicie `Remove`, `RemoveAt` i `RemoveRange`. Pierwsza z nich (`Remove`) usuwa początkowe wystąpienie wartości podanej jako parametr. Metoda `RemoveAt` usuwa element o indeksie równym wartości parametru, a ostatnia z metod (`RemoveRange`) umożliwia usunięcie podanej liczby elementów, zaczynając od określonego indeksu. Oprócz tego do usunięcia wszystkich elementów używa się metody `Clear`.

Z innych metod warto wspomnieć metodę `Reverse`, odwracającą kolejność elementów listy tablicowej, oraz `ToArray`, która zwraca tablicę zawierającą wszystkie elementy przechowywane w instancji `ArrayList`.

> Więcej informacji na temat klasy `ArrayList` można uzyskać pod adresem *https://msdn.microsoft.com/pl-pl/library/system.collections.arraylist.aspx*.

Lista generyczna

Jak widać, klasa `ArrayList` zawiera szeroki zakres funkcji, ale ma znaczący mankament — nie jest listą silnie typowaną. Aby korzystać z listy silnie typowanej, można użyć klasy generycznej `List` — kolekcji, której rozmiar daje się zwiększać albo zmniejszać zależnie od potrzeb.

Klasa generyczna `List` zawiera dużo właściwości i metod bardzo użytecznych podczas opracowywania aplikacji przechowujących dane. Warto zauważyć, że wiele składowych ma te same nazwy co w klasie `ArrayList`, znajdziemy w niej na przykład właściwości `Count` i `Capacity` oraz metody `Add`, `AddRange`, `Clear`, `Contains`, `IndexOf`, `Insert`, `InsertRange`, `LastIndexOf`, `Remove`, `RemoveAt`, `RemoveRange`, `Reverse` i `ToArray`. Konkretny element listy również pozyskuje się za pomocą indeksu i operatora `[]`.

Oprócz podanych funkcji można także używać wszechstronnego zestawu metod rozszerzających z przestrzeni nazw `System.Linq`, na przykład do znajdowania minimalnej lub maksymalnej wartości (`Min` i `Max`), obliczania średniej (`Average`), porządkowania listy w kolejności rosnącej lub

Kup ksi k Pole ksi k

malejącej (OrderBy i OrderByDescending) oraz sprawdzania, czy wszystkie elementy spełniają warunek (All). To oczywiście nie wszystkie funkcje dostępne przy tworzeniu aplikacji w języku C# z wykorzystaniem list generycznych.

> Więcej informacji na temat klasy generycznej List można uzyskać pod adresem *https://msdn.microsoft.com/library/6sh2ey19.aspx*.

Przyjrzyjmy się dwóm przykładom pokazującym praktyczne wykorzystanie listy generycznej.

Przykład — średnia wartość

Pierwszy przykład korzysta z klasy generycznej List do przechowywania wartości zmiennoprzecinkowych (typu double) wprowadzonych przez użytkownika. Po wpisaniu liczby jest obliczana i pokazywana w konsoli średnia. Program przerywa działanie, jeśli użytkownik wprowadził niepoprawną wartość.

Kod metody Main w klasie Program wygląda następująco:

```
List<double> numbers = new List<double>();
do
{
    Console.Write("Wprowadź liczbę: ");
    string numberString = Console.ReadLine();
    if (!double.TryParse(numberString, NumberStyles.Float,
        new NumberFormatInfo(), out double number))
    {
        break;
    }
    numbers.Add(number);
    Console.WriteLine($"Średnia wartość: {numbers.Average()}");
}
while (true);
```

Na początku tworzone jest wystąpienie klasy List. Następnie w nieskończonej pętli (do while) program czeka, aż użytkownik wprowadzi liczbę. Jeśli jest ona poprawna, wprowadzoną wartość dodaje do listy (wywołując metodę Add), oblicza średnią wartość elementów listy (wywołując metodę Average) i pokazuje ją w konsoli.

W rezultacie otrzymuje się dane wyjściowe podobne do tych:

```
Wprowadź liczbę: 10.5
Średnia wartość: 10.5 (...)
Wprowadź liczbę: 1.5
Średnia wartość: 4.875
```

W tym przykładzie przyjrzałeś się, jak używać listy zawierającej wartości typu double. Czy jednak może ona również przechowywać wystąpienia klas zdefiniowanych przez użytkownika? Oczywiście! W następnym przykładzie zobaczysz, jak to zrobić.

Przykład — lista osób

Drugi przykład dotyczący klasy `List` pokazuje, jak użyć tej struktury danych do utworzenia bardzo prostej bazy danych osób. Przechowywane będzie imię, kraj i wiek każdej z nich. Po uruchomieniu programu do listy dodaje się dane kilku osób. Potem dane są sortowane (za pomocą wyrażenia LINQ) i prezentowane w konsoli.

Rozpocznijmy od deklaracji klasy `Person`, pokazanej na poniższym listingu:

```
public class Person {
public string Name { get; set; }
public int Age { get; set; }
public CountryEnum Country { get; set; } }
```

Klasa ta zawiera trzy publiczne właściwości, a mianowicie `Name`, `Age` i `Country`. Warto odnotować, że właściwość `Country` jest typu `CountryEnum`, definiującego trzy stałe, to znaczy `PL` (Polska), `UK` (Wielka Brytania) i `DE` (Niemcy), co pokazuje poniższy kod:

```
public enum CountryEnum {
    PL,
    UK,
    DE
 }
```

Kolejny fragment kodu powinien trafić do metody `Main` klasy `Program`. Tworzy ona nowe wystąpienie klasy `List` i dodaje dane kilku osób o różnych imionach, kraju pochodzenia i wieku, co pokazano poniżej:

```
List<Person> people = new List<Person>();
 people.Add(new Person() { Name = "Marcin",
    Country = CountryEnum.PL, Age = 29 });
 people.Add(new Person() { Name = "Sabine",
    Country = CountryEnum.DE, Age = 25 }); (...)
 people.Add(new Person() { Name = "Anna",
    Country = CountryEnum.PL, Age = 31 });
```

W następnym wierszu za pomocą wyrażenia LINQ sortuje się listę według imion w kolejności rosnącej, a uzyskany wynik zamienia się z powrotem na listę:

```
List results = people.OrderBy(p => p.Name).ToList();
```

Potem można z łatwością przejść po wynikach za pomocą pętli `foreach`:

```
foreach (Person person in results)
{
    Console.WriteLine($"{person.Name} (lat {person.Age}) z {person.Country}.");
}
```

Po uruchomieniu program pokazuje następujący wynik:

```
Marcin (lat 29) z PL. (...)
Sabine (lat 25) z DE.
```


Kup ksi k   Pole ksi k

To wszystko! Omówmy teraz nieco szerzej wyrażenia LINQ, których można użyć do porządkowania elementów, ale również do ich filtrowania w oparciu o zadane kryteria, i nie tylko.

Przyjrzyjmy się na przykład następującemu zapytaniu używającemu **składni metody** (ang. *method syntax*):

```
List names = people.Where(p => p.Age <= 30)
    .OrderBy(p => p.Name)
    .Select(p => p.Name)
    .ToList();
```

Wybiera ona imiona (klauzula `Select`) wszystkich osób, których wiek jest równy `30` lat albo mniej (klauzula `Where`), uporządkowane w kolejności imion (klauzula `OrderBy`). Następnie wykonywane jest zapytanie, a jego wyniki są zwracane w postaci listy.

To samo można zrobić, używając pokazanej poniżej **składni zapytania** (ang. *query syntax*) w połączeniu z wywołaniem metody `ToList`:

```
List names = (from p in people
    where p.Age <= 30
    orderby p.Name
    select p.Name).ToList();
```

W tym podrozdziale przyjrzałeś się, jak za pomocą klasy `ArrayList` oraz klasy generycznej `List` przechowywać dane w kolekcjach o dynamicznie zmienianym rozmiarze. Nie jest to jednak koniec zagadnień związanych z listami w tym rozdziale. Czy jesteś gotowy poznać kolejną strukturę danych, przechowującą posortowane elementy? Jeśli tak, przejdź do kolejnego podrozdziału, poświęconego listom uporządkowanym.

Listy uporządkowane

W tym rozdziale nauczyłeś się już przechowywać dane w tablicach i listach, ale czy wiesz, że możesz korzystać ze struktury danych zapewniającej uporządkowanie elementów? Jeśli nie, poznaj klasę generyczną `SortedList` (z przestrzeni nazw `System.Collections.Generic`), która jest kolekcją **par klucz-wartość** (ang. *key-value pairs*) uporządkowanych według kluczy, niewymagającą sortowania przez użytkownika. Warto wspomnieć, że wszystkie klucze muszą być niepowtarzalne i nie mogą mieć wartości `null`.

Aby dodać element do kolekcji, używa się po prostu metody `Add`, a żeby usunąć określony element — metody `Remove`. Spośród wielu innych warto wspomnieć jeszcze o metodach `ContainsKey` i `ContainsValue`, sprawdzających, czy kolekcja zawiera element o danym kluczu albo wartości, oraz o metodach `IndexOfKey` i `IndexOfValue`, zwracających indeks elementu o podanym kluczu lub wartości. Ponieważ lista uporządkowana przechowuje pary klucz-wartość, zapewnia dostęp do właściwości `Keys` i `Values`. Poszczególne klucze i wartości można z łatwością pozyskać za pomocą indeksu i operatora `[]`.

Kup ksi k

Pole ksi k

Więcej informacji na temat klasy generycznej `SortedList` można uzyskać pod adresem *https://msdn.microsoft.com/library/ms132319.aspx*.

Po tym krótkim wprowadzeniu spójrzmy na przykład pokazujący, jak korzystać z tej struktury danych, a także wskazujący pewne znaczące różnice w kodzie w porównaniu z opisaną wcześniej klasą `List`.

Przykład — książka adresowa

W tym przykładzie użyto klasy `SortedList` do stworzenia bardzo prostej książki adresowej, uporządkowanej według imion. Przechowuje ona następujące dane każdej osoby: `Name` (imię), `Age` (wiek) i `Country` (kraj). Deklarację klasy `Person` przedstawia poniższy kod:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public CountryEnum Country { get; set; }
}
```

Właściwości `Country` można nadać wartość jednej ze stałych z wyliczenia `CountryEnum`:

```
public enum CountryEnum {
    PL,
    UK,
    DE
 }
```

Najbardziej interesujący fragment kodu znajduje się w metodzie `Main` klasy `Program`. Tworzone jest tam nowe wystąpienie klasy generycznej `SortedList` o określonych typach kluczy i wartości, mianowicie `string` i `Person`, co pokazuje poniższy kod:

```
SortedList<string, Person> people =
    new SortedList<string, Person>();
```

Potem do listy uporządkowanej można z łatwością dodać dane za pomocą metody `Add`, przekazując dwa parametry, to znaczy klucz (czyli imię) i wartość (czyli wystąpienie klasy `Person`), co pokazuje poniższy listing:

```
people.Add("Marcin", new Person() { Name = "Marcin",
    Country = CountryEnum.PL, Age = 29 });
people.Add("Sabine", new Person() { Name = "Sabine",
    Country = CountryEnum.DE, Age = 25 }); (...)
people.Add("Anna", new Person() { Name = "Anna",
    Country = CountryEnum.PL, Age = 31 });
```

Kiedy wszystkie dane zostaną zapisane w kolekcji, można bez problemu przejść przez jej elementy (pary klucz-wartość) za pomocą pętli `foreach`. Warto wspomnieć, że w pętli użyto zmiennej typu `KeyValuePair`. Aby zatem uzyskać dostęp do klucza i wartości, należy skorzystać z, odpowiednio, właściwości `Key` i `Value`, tak jak poniżej:

Kup ksi k Pole ksi k

```
foreach (KeyValuePair person in people)
{
    Console.WriteLine($"{person.Value.Name} (lat {person.Value.Age})
    ↪z {person.Value.Country}.");
}
```

Po uruchomieniu programu w konsoli pokaże się następujący wynik:

```
Anna (lat 31) z PL. (...)
Marcin (lat 29) z PL. (...)
Sabine (lat 25) z DE.
```

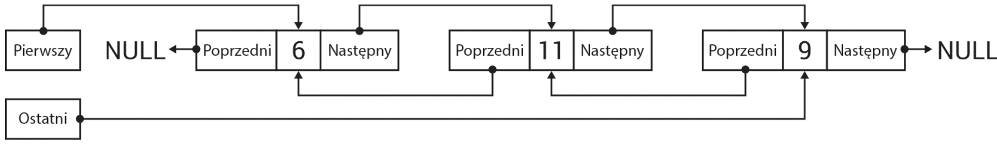
Jak widać, kolekcja jest automatycznie sortowana według imion użytych jako klucze listy uporządkowanej. Należy jednak pamiętać, że klucze nie mogą się powtarzać, w tym przykładzie więc nie można dodać więcej niż jednej osoby o tym samym imieniu.

Listy wiązane

Korzystając z klasy generycznej `List`, można z łatwością za pomocą indeksów uzyskać dostęp do konkretnych elementów kolekcji, ale w jaki sposób, po pobraniu jednego elementu kolekcji, przejść do następnego? Czy to w ogóle możliwe? Można rozważyć użycie w tym celu metody `IndexOf`, pozwalającej uzyskać indeks elementu. Niestety zwraca ona indeks pierwszego wystąpienia danej wartości w kolekcji, w tym przypadku nie zawsze działa więc zgodnie z oczekiwaniami.

Wspaniale byłoby mieć pewnego rodzaju *wskaźnik* do następnego elementu, co pokazuje poniższy rysunek:

Dzięki temu podejściu można łatwo przejść od jednego elementu do następnego, korzystając z właściwości `Next`. Taka struktura nosi nazwę **listy jednokierunkowej** (ang. *single-linked list*). Czy da się ją jednak jeszcze bardziej rozszerzyć, dodając właściwość `Previous`, aby można było przemieszczać się do przodu i do tyłu? Oczywiście! Taką strukturę danych, która nosi nazwę **listy dwukierunkowej** (ang. *double-linked list*), przedstawia poniższy rysunek:



Jak widać, lista dwukierunkowa zawiera właściwość `First`, która wskazuje jej pierwszy element. Każdy element ma dwie właściwości, wskazujące poprzedni i następny element (odpowiednio `Previous` i `Next`). Jeśli nie ma poprzedniego elementu, właściwość `Previous` ma wartość `null`.

Kup ksi k Pole ksi k

Analogicznie jeśli nie ma następnego elementu, właściwość `Next` ma wartość `null`. Lista dwukierunkowa zawiera ponadto właściwość `Last` wskazującą ostatni element. Jeśli w liście nie ma elementów, obie właściwości `First` i `Last` mają wartość `null`.

Czy taką strukturę danych musisz zaimplementować samodzielnie, aby jej używać w swoich aplikacjach w języku C#? Na szczęście nie, ponieważ jest dostępna jako klasa generyczna `LinkedList` w przestrzeni nazw `System.Collections.Generic`.

Tworząc wystąpienie tej klasy, należy podać parametr wskazujący typ pojedynczego elementu listy, na przykład `int` albo `string`. Niemniej jednak pojedynczy węzeł nie jest po prostu zmienną typu `int` ani `string`, ponieważ gdyby tak było, brakowałoby dostępu do dodatkowych właściwości listy dwukierunkowej, takich jak `Previous` albo `Next`. Aby rozwiązać ten problem, każdy węzeł jest wystąpieniem klasy generycznej `LinkedListNode`, na przykład `LinkedListNode<int>` lub `LinkedListNode<string>`.

Wymagane jest pewne dodatkowe objaśnienie metod dodających nowe węzły do listy dwukierunkowej. Używa się do tego metod ustawiających, a mianowicie:

- `AddFirst`: w celu dodania elementu na początek listy,
- `AddLast`: w celu dodania elementu na koniec listy,
- `AddBefore`: w celu dodania elementu przed podanym węzłem listy,
- `AddAfter`: w celu dodania elementu po podanym węźle listy.

Wszystkie te metody zwracają wystąpienie klasy `LinkedListNode`. Są także inne metody, takie jak `Contains`, sprawdzająca, czy w liście istnieje podana wartość, `Clear`, usuwająca z listy wszystkie elementy, czy też `Remove`, pozwalająca usunąć węzeł z listy.

> Więcej informacji na temat klasy generycznej `LinkedList` można uzyskać pod adresem *https://msdn.microsoft.com/library/he2s3bh7.aspx*.

Po tym krótkim wprowadzeniu jesteś już pewnie gotowy, aby przyjrzeć się przykładowi pokazującemu, jak zastosować listę dwukierunkową w postaci klasy `LinkedList` w praktyce.

Przykład — czytnik książki

Jako przykład przygotujesz prostą aplikację umożliwiającą czytanie książki ze zmianą stron. Użytkownik po naciśnięciu klawisza *N* przechodzi do następnej strony (jeśli istnieje), a po naciśnięciu *P* wraca do poprzedniej (jeśli jest dostępna). Treść bieżącej strony oraz jej numer powinny być pokazane w konsoli, tak jak na poniższym rysunku:

Kup ksi k

Pole ksi k

```
C:\WINDOWS\system32\cmd.exe
                                    - 3 -

Duża liczba aplikacji korzysta z rozmaitych danych, takich jak posty, zdjęcia lub po prost
u ustawienia, pozwalające użytkownikom dopasować aplikację do swoich potrzeb. Dane można o
czywiście przechowywać w różny sposób, na przykład w plikach tekstowych lub binarnych albo
 w bazach danych. Nawet zawartość prostego pliku tekstowego może być różnie zdefiniowana,
na przykład w postaci zwykłego tekstu albo w formacie Extensible Markup Language (XML) lub
 JavaScript Object Notation (JSON). Wybór odpowiedniego formatu przechowywania danych w ap
likacji jest ważnym zadaniem, wpływającym na rozwój i utrzymanie projektu oraz na wydajnoś
ć zapisywania i odczytywania danych.

Cytat z książki Marcina Jamry "Windows Application Development Cookbook",
Packt Publishing, 2016 r.

< POPRZEDNIA [P]                                                        [N] NASTĘPNA >
```

Rozpocznijmy od poniższej deklaracji klasy `Page`:

```
public class Page
{
    public string Content { get; set; }
}
```

Klasa ta reprezentuje jedną stronę i zawiera właściwość `Content`. W metodzie `Main` klasy `Program` trzeba utworzyć kilka wystąpień klasy `Page`, reprezentujących sześć stron książki, co pokazuje poniższy listing:

```
Page pageFirst = new Page() { Content = "W dzisiejszych czasach (...)" };
Page pageSecond = new Page() { Content = "Opracowywanie (...)" };
Page pageThird = new Page() { Content = "Duża liczba (...)" };
Page pageFourth = new Page() { Content = "Czy wiesz (...)" };
Page pageFifth = new Page() { Content = "Opracowując (...)" };
Page pageSixth = new Page() { Content = "Czy możesz (...)" };
```

Po utworzeniu wystąpień przejdźmy do konstruowania listy wiązanej, korzystając z kilku metod dodających węzły, pokazanych na poniższym listingu:

```
LinkedList<Page> pages = new LinkedList<Page>();
pages.AddLast(pageSecond);
LinkedListNode<Page> nodePageFourth = pages.AddLast(pageFourth);
pages.AddLast(pageSixth);
pages.AddFirst(pageFirst);
pages.AddBefore(nodePageFourth, pageThird);
pages.AddAfter(nodePageFourth, pageFifth);
```

W pierwszym wierszu tworzona jest nowa lista, a potem wykonywane są następujące operacje:

- dodanie danych drugiej strony na końcu listy (`[2]`),
- dodanie danych czwartej strony na końcu listy (`[2, 4]`),
- dodanie danych szóstej strony na końcu listy (`[2, 4, 6]`),
- dodanie danych pierwszej strony na początku listy (`[1, 2, 4, 6]`),
- dodanie danych trzeciej strony przed węzłem czwartej (`[1, 2, 3, 4, 6]`),
- dodanie danych piątej strony po węźle czwartej (`[1, 2, 3, 4, 5, 6]`).

Kup ksi k   Pole ksi k

Kolejny fragment kodu odpowiada za prezentację strony w konsoli oraz za nawigację pomiędzy stronami po naciśnięciu odpowiednich klawiszy. Kod ten wygląda następująco:

```
LinkedListNode<Page> current = pages.First; int number = 1;
while (current != null)
{
    Console.Clear();
    string numberString = $"- {number} -";
    int leadingSpaces = (90 - numberString.Length) / 2;
    Console.WriteLine(numberString.PadLeft(leadingSpaces
        + numberString.Length));
    Console.WriteLine();

    string content = current.Value.Content;
    for (int i = 0; i < content.Length; i += 90)
    {
        string line = content.Substring(i);
        line = line.Length > 90 ? line.Substring(0, 90) : line;
        Console.WriteLine(line);
    }

    Console.WriteLine();
    Console.WriteLine($"Cytat z książki Marcina Jamry \"Windows Application
    ↪Development Cookbook\",{Environment.NewLine}Packt Publishing, 2016 r.");

    Console.WriteLine();
    Console.Write(current.Previous != null
        ? "< POPRZEDNIA [P]" : GetSpaces(16));
    Console.Write(current.Next != null
        ? "[N] NASTĘPNA >".PadLeft(74) : string.Empty);
    Console.WriteLine();

    switch (Console.ReadKey(true).Key)
    {
        case ConsoleKey.N:
            if (current.Next != null)
            {
                current = current.Next;
                number++;
            }
            break;
        case ConsoleKey.P:
            if (current.Previous != null)
            {
                current = current.Previous;
                number--;
            }
            break;
        default:
            return;
    }
}
```


Kup ksi k Pole ksi k

Przydałoby się wyjaśnić ten listing. W pierwszym wierszu zmiennej `current` nadaje się wartość pierwszego węzła listy wiązanej. Ogólnie rzecz biorąc, zmienna `current` reprezentuje stronę aktualnie pokazywaną w konsoli. Następnie numerowi strony (zmiennej `number`) nadaje się początkową wartość `1`. Jednak najbardziej interesujący i skomplikowany fragment kodu jest ukazany w pętli `while`.

Wewnątrz pętli jest czyszczona bieżąca zawartość konsoli i odpowiednio formatowany jest ciąg z numerem strony do wyświetlenia. Przed numerem i po nim dodawane są znaki `-`. W celu wyśrodkowania ciągu w poziomie wstawiane są ponadto początkowe spacje (za pomocą metody `PadLeft`).

Następnie treść strony jest dzielona na wiersze niemające więcej niż 90 znaków i wypisywana do konsoli. Do podzielenia ciągu używa się metody `Substring` i właściwości `Length`. W podobny sposób w konsoli pokazywana jest dodatkowa informacja (o książce, z której pochodzą cytaty). Warto wspomnieć o właściwości `Environment.NewLine`, która wstawia podział wiersza w określonym miejscu ciągu. Następnie pokazywane są napisy `POPRZEDNIA` i `NASTĘPNA`, jeśli dostępna jest poprzednia lub następna strona.

W kolejnej części kodu program czeka, aż użytkownik naciśnie dowolny klawisz, ale nie pokazuje go w konsoli (dzięki przesłaniu jako parametru wartości `true`). Gdy użytkownik naciśnie klawisz *N*, za pomocą właściwości `Next` zmienna `current` jest ustawiana na następny węzeł. Ta operacja nie jest oczywiście wykonywana, jeśli następna strona jest niedostępna. W podobny sposób obsługiwany jest klawisz *P*, który przenosi użytkownika na poprzednią stronę. Warto wspomnieć, że wraz ze zmianą wartości zmiennej `current` modyfikowany jest numer strony (zmienna `number`).

Na koniec zobaczmy kod pomocniczej metody `GetSpaces`:

```
private static string GetSpaces(int number)
{
    string result = string.Empty;
    for (int i = 0; i < number; i++)
    {
        result += " ";
    }
    return result;
}
```

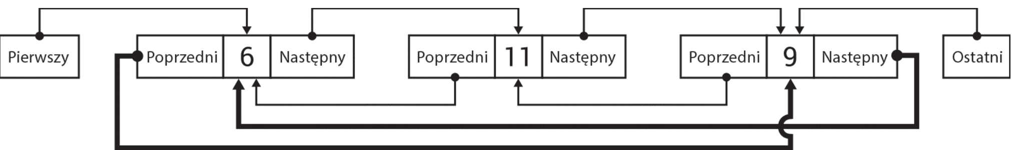
Ta metoda jedynie przygotowuje i zwraca zmienną typu `string` o określonej liczbie spacji.

Listy cykliczne

W poprzednim podrozdziale dowiedziałeś się o listach dwukierunkowych. Jak widać, implementacja tej struktury danych pozwala nawigować między węzłami za pomocą właściwości `Previous` i `Next`. Niemniej jednak właściwość `Previous` pierwszego węzła ma wartość `null`, podobnie jak właściwość `Next` ostatniego węzła. Czy wiesz, że możesz łatwo rozwinąć tę metodę i stworzyć **listę cykliczną** (ang. *circular-linked list*)?

Kup ksi k   Pole ksi k

Taką strukturę danych pokazuje poniższy rysunek:



W tym przypadku właściwość `Previous` pierwszego węzła kieruje do ostatniego, a właściwość `Next` ostatniego węzła kieruje do pierwszego. Taka struktura danych w pewnych przypadkach bywa użyteczna. Przekonasz się o tym, opracowując wzięty z życia przykład.

> Warto wspomnieć, że do przechodzenia między węzłami niekoniecznie trzeba używać właściwości. Można je zamienić na metody, co zobaczysz w przykładzie w następnym punkcie rozdziału.

Implementacja

Po krótkim wprowadzeniu w temat list cyklicznych czas przyjrzeć się ich implementacji. Zacznijmy od poniższego kodu:

```
public class CircularLinkedList<T> : LinkedList<T>
{
    public new IEnumerator GetEnumerator()
    {
        return new CircularLinkedListEnumerator<T>(this);
    }
}
```

Kod ten pokazuje, że listę cykliczną można zaimplementować jako klasę generyczną rozszerzającą typ `LinkedList`. Warto wspomnieć, że implementacja metody `GetEnumerator` korzysta z klasy `CircularLinkedListEnumerator`. Dzięki jej utworzeniu będzie można przechodzić w nieskończoność przez wszystkie elementy listy cyklicznej za pomocą pętli `foreach`.

Kod klasy `CircularLinkedListEnumerator` wygląda następująco:

```
public class CircularLinkedListEnumerator<T> : IEnumerator<T>
{
    private LinkedListNode<T> _current;
    public T Current => _current.Value;
    object IEnumerator.Current => Current;

    public CircularLinkedListEnumerator(LinkedList<T> list)
    {
        _current = list.First;
    }

    public bool MoveNext()
    {
        if (_current == null)
```


Kup ksi k
Pole ksi k

```
            {
                return false;
            }

            _current = _current.Next ?? _current.List.First;
            return true;
        }

        public void Reset()
        {
            _current = _current.List.First;
        }

        public void Dispose() { }
    }
```

Klasa `CircularLinkedListEnumerator` implementuje interfejs `IEnumerator`. Deklaruje ona pole o dostępie `private` reprezentujące podczas iteracji po liście bieżący węzeł (`_current`). Zawiera również dwie właściwości, to znaczy `Current` i `IEnumerator.Current`, które są wymagane przez interfejs `IEnumerator`. Konstruktor jedynie nadaje wartość zmiennej `_current` w oparciu o wystąpienie klasy `LinkedList`, przekazane jako parametr.

Jedną z najważniejszych części kodu jest metoda `MoveNext`. Zatrzymuje ona iterację, jeśli zmienna `_current` ma wartość `null`, to znaczy, jeśli lista nie ma elementów. W przeciwnym przypadku zmienia bieżący element na następny albo na pierwszy węzeł listy, jeśli następny węzeł jest niedostępny. Metoda `Reset` jedynie nadaje polu `_current` wartość pierwszego węzła listy.

Na koniec należy stworzyć dwie metody rozszerzeń pozwalające przejść do pierwszego elementu podczas próby pozyskania elementu następującego po ostatniej pozycji listy oraz przejść do ostatniego elementu podczas próby pozyskania elementu poprzedzającego pierwszą pozycję listy. Aby uprościć implementację, funkcje te będą dostępne jako metody `Next` i `Previous` zamiast pokazanych na poprzednim rysunku właściwości `Next` i `Previous`. Oto ich kod:

```
    public static class CircularLinkedListExtensions {
        public static LinkedListNode<T> Next<T>(
            this LinkedListNode<T> node)
        {
            if (node != null && node.List != null)
            {
                return node.Next ?? node.List.First;
            }
            return null;
        }
        public static LinkedListNode<T> Previous<T>(
            this LinkedListNode<T> node)
        {
            if (node != null && node.List != null)
            {
                return node.Previous ?? node.List.Last;
            }
            return null;
        }
    }
```


Kup ksi k

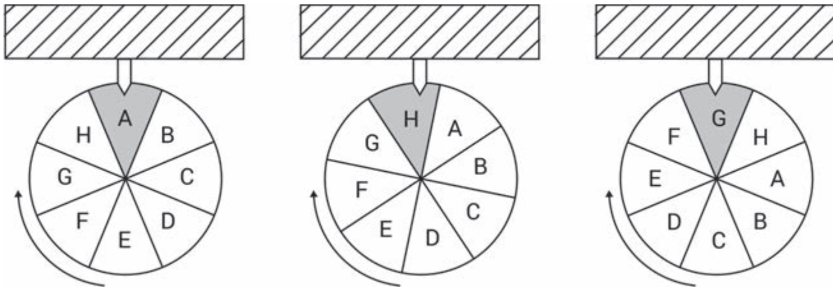
Pole ksi k

Pierwsza metoda rozszerzenia o nazwie Next sprawdza, czy węzeł istnieje i czy lista jest dostępna. Jeśli tak, zwraca wartość właściwości Next węzła (jeśli ta nie jest równa null) albo zwraca referencję do pierwszego elementu listy za pomocą właściwości First. Metoda Previous działa w podobny sposób.

To wszystko! Właśnie skończyłeś implementować w języku C# listę cykliczną, której będziesz mógł używać w rozmaitych aplikacjach. W jaki sposób? Przyjrzyj się poniższemu przykładowi korzystającemu z tej struktury danych.

Przykład — zakręć kołem

Ten przykład symuluję grę, w której użytkownik kręci kołem z przypadkową prędkością. Koło obraca się coraz wolniej, aż w końcu zatrzymuje się. Użytkownik może potem znowu nim zakręcić, zaczynając z pozycji, na której się zatrzymało, co obrazuje poniższy rysunek:



Przejdźmy do pierwszej części kodu metody Main w klasie Program:

```
CircularLinkedList<string> categories =
    new CircularLinkedList<string>();
categories.AddLast("Sport");
categories.AddLast("Kultura");
categories.AddLast("Historia");
categories.AddLast("Geografia");
categories.AddLast("Ludzie");
categories.AddLast("Technologia");
categories.AddLast("Przyroda");
categories.AddLast("Nauka");
```

Na początku tworzone jest nowe wystąpienie klasy CircularLinkedList, reprezentującej listę cykliczną o elementach typu string. Potem dodawanych jest osiem wartości, a mianowicie Sport, Kultura, Historia, Geografia, Ludzie, Technologia, Przyroda i Nauka.

Najważniejsze operacje wykonuje poniższy fragment kodu:

```
Random random = new Random();
int totalTime = 0;
int remainingTime = 0;
foreach (string category in categories)
{
```


Kup ksi k   Pole ksi k

```
        if (remainingTime <= 0)
        {
            Console.WriteLine("Naciśnij klawisz [Enter], aby zacząć,
            ↪lub dowolny inny, aby wyjść.");
            switch (Console.ReadKey().Key)
            {
                case ConsoleKey.Enter:
                    totalTime = random.Next(1000, 5000);
                    remainingTime = totalTime;
                    break;
                default:
                    return;
            }
        }

        int categoryTime = (-450 * remainingTime) / (totalTime - 50)
             + 500 + (22500 / (totalTime - 50));
        remainingTime -= categoryTime;
        Thread.Sleep(categoryTime);

        Console.ForegroundColor = remainingTime <= 0
            ? ConsoleColor.Red : ConsoleColor.Gray;
        Console.WriteLine(category);
        Console.ForegroundColor = ConsoleColor.Gray;
    }
```

Najpierw deklarowane są trzy zmienne, przechowujące wylosowaną wartość (`random`), całkowity wylosowany czas kręcenia się koła w milisekundach (`totalTime`) oraz czas pozostałych obrotów koła, również w milisekundach (`remainingTime`).

Następnie pętla `foreach` przechodzi przez wszystkie elementy listy cyklicznej. Jeśli w takiej pętli nie będzie instrukcji `break` ani `return`, to z powodu natury listy cyklicznej pętla będzie się wykonywać bez końca. Po dojściu do ostatniego elementu w kolejnej iteracji pobrana zostanie automatycznie pierwsza pozycja z listy.

W pętli sprawdza się pozostały czas. Jeśli jest on mniejszy od zera lub równy zero, co oznacza, że koło się zatrzymało albo jeszcze nie zaczęło się obracać, pokazywany jest komunikat, a program czeka na naciśnięcie klawisza *Enter*. W takiej sytuacji konfigurowana jest nowa operacja kręcenia kołem, następuje losowanie całkowitego czasu obrotów i ustawienie pozostałego czasu. Jeśli użytkownik naciśnie inny klawisz, program przerywa działanie.

W następnym kroku obliczany jest czas jednej iteracji pętli. Formuła zapewnia krótsze czasy na początku (koło kręci się szybciej), a dłuższe na końcu (koło kręci się wolniej). Pozostały czas jest zmniejszany, a program czeka przez określoną liczbę milisekund, używając metody `Sleep`.

Na koniec, podczas przedstawiania ostatecznego wyniku, kolor pierwszego planu jest zmieniany na czerwony, a kategoria, którą aktualnie wskazuje koło, jest prezentowana w konsoli.

Kup ksi k Pole ksi k

Po uruchomieniu aplikacji można uzyskać następujący rezultat:

```
Naciśnij klawisz [Enter], aby zacząć, lub dowolny inny, aby wyjść.
Kultura
Historia
Geografia (...)
Kultura
Historia
Naciśnij klawisz [Enter], aby zacząć, lub dowolny inny, aby wyjść.
Geografia (...)
Przyroda
Nauka (...)
Ludzie
Technologia
Naciśnij klawisz [Enter], aby zacząć, lub dowolny inny, aby wyjść.
```

Zakończyłeś już przykład wykorzystujący listę cykliczną. Jest ona jedną ze struktur danych opisanych w tym rozdziale. Jeśli chcesz, przejdź do krótkiego podsumowania zdobytych informacji.

Podsumowanie

Tablice i listy to jedne ze struktur danych najczęściej używanych przy opracowywaniu różnego rodzaju aplikacji. Niemniej jednak temat ten nie jest tak prosty, jak się wydaje, ponieważ nawet wśród tablic wyróżnia się kilka wariantów, to znaczy tablice jednowymiarowe, wielowymiarowe i nieregularne, zwane również tablicami tablic.

Jeśli chodzi o listy, różnice są jeszcze bardziej widoczne, co można zobaczyć na przykładzie list prostych, generycznych, uporządkowanych, jednokierunkowych, dwukierunkowych i cyklicznych. Na szczęście dostępna jest wbudowana implementacja listy tablicowej, a także generycznej, uporządkowanej i dwukierunkowej. Poza tym listę dwukierunkową da się dość łatwo rozszerzyć, aby działała jak lista cykliczna. Można zatem wykorzystać możliwości, jakie dają odpowiednie struktury, bez znacznego nakładu pracy.

Być może omówione typy struktur danych wyglądają na dość skomplikowane, ale w tym rozdziale zobaczyłeś szczegółowe opisy poszczególnych struktur oraz implementację przykładów w języku C#. Powinny one ułatwić sprawę i mogą stanowić podstawę Twoich przyszłych projektów.

Czy jesteś gotowy, aby poznać inne struktury danych? Jeśli tak, przejdź do następnego rozdziału i poczytaj o stosach i kolejkach!

Kup ksi k
Pole ksi k

Kup ksi k

Pole ksi k

Skorowidz

A

algorytm
  Dijkstry, 207
  Kruskala, 193
  Prima, 196
algorytmy sortowania, 45

B

binarne drzewo poszukiwań, BST, 139

C

ciągi, 18

D

debugowanie, 30
delegaty, 21
drzewa, 123–125, 133, 143, 221
  AVL, 156
  czerwono-czarne, 159
  implementacja, 157
  binarne, 129
    zupełne, 162
  BST
    implementacja, 142
    usuwanie, 145
    wizualizacja, 149
    wstawianie, 144
    wyszukiwanie, 143
  dwumianowe, 165
  MST, 193
  niezrównoważone, 156
  poszukiwań, 139
  RBT
    implementacja, 160
  rozpinające, 192
  zrównoważone, 156
działania na zbiorach, 114

F

FIFO, 82
funkcje
  drzew RBT, 160
  skrótu, 100

G

grafy, 169, 223
  cykle, 170
  implementacja, 178
  jednokierunkowe, 171
  krawędzie, 170, 179
  najkrótsza ścieżka, 207
  nieskierowane, 171, 184
  nieważone, 171, 184
  pętle, 170
  przeszukiwanie w głąb, 186
  przeszukiwanie wszerz, 189
  skierowane, 171, 185
  wagi, 171
  ważone, 171, 185
  węzły, 170, 178
  wierzchołki, 170
  zastosowania, 172

Kup ksi k   Pole ksi k

H

haszowanie, 100
hierarchia identyfikatorów, 126

I

IDE, 22
  instalacja, 22
  konfiguracja środowiska, 22
  okno bezpośrednie, 32
implementacja
  drzewa, 124
    AVL, 157
    binarnego, 132
    BST, 142
    RBT, 160
  grafu, 178
  kopca binarnego, 163
  list cyklicznych, 67
interfejsy, 20

J

język programowania, 14

K

klasa, 19
  Hashtable, 100
  Object, 19
klasyfikacja struktur danych, 215
klucz, 100
kolejki, 82, 219
  priorytetowe, 92
kolorowanie węzłów, 203
kopce, 222
  binarne, 162
  dwumianowe, 165
  maksymalne, 162
  minimalne, 162
kopiec Fibonacciego, 167
krawędzie
  nieskierowane, 171, 184
  nieważone, 184
  skierowane, 171, 185
  ważone, 185

L

lewe dziecko, 129
LIFO, 74
listy, 55, 218
  cykliczne, 66
  dwukierunkowe, 62
  generyczne, 57
  jednokierunkowe, 62
  sąsiedztwa, 173
  tablicowe, 55
  uporządkowane, 60
  wiązane, 62

M

macierz sąsiedztwa, 173, 175
mapa z haszowaniem, 99
menedżer pakietów NuGet, 163
metoda GetMinimumWeightIndex, 199
minimalne drzewo rozpinające, MST, 192

N

najkrótsza ścieżka, 207

O

odczytywanie z wejścia, 26
okno bezpośrednie, 32

P

prawe dziecko, 129
przechodzenie drzewa, 129
przejście
  poprzeczne, 129
  wsteczne, 129
  wzdłużne, 129
przeszukiwanie
  w głąb, 186
  wszerz, 189
przykład
  baseny, 117
  biuro telefonicznej obsługi, 94
  czytnik książki, 63
  dane użytkownika, 107
  definicje, 110
  funkcje drzew RBT, 160
  hierarchia identyfikatorów, 126

Kup ksi k

Pole ksi k

kabel telekomunikacyjny, 200
krawędzie nieskierowane, 184
krawędzie skierowane, 185
książka adresowa, 61
książka telefoniczna, 101
kupony, 115
lista osób, 59
mapa gry, 38, 210
mapa województw, 205
nazwy miesięcy, 36
odwracanie wyrazów, 75
prosty quiz, 136
roczny plan transportu, 42
sortowanie przez kopcowanie, 164
struktura przedsiębiorstwa, 127
średnia wartość, 58
tabliczka mnożenia, 37
telefoniczne biuro obsługi, 84, 88
usuwanie duplikatów, 121
Wieże Hanoi, 75
wizualizacja drzewa BST, 149
wyszukiwanie produktu, 105
zakręć kołem, 69
zrównoważenie drzewa, 158

R

reprezentacja, 173

S

słowniki, 99, 104, 220
uporządkowane, 109
słowo kluczowe const, 17
sortowanie
bąbelkowe, 50
przez kopcowanie, 164
przez wstawianie, 48
przez wybieranie, 45
szybkie, 52
stałe, 17
stosy, 73, 219
struktury danych, 16, 215
o dostępie sekwencyjnym, 73
o dostępie swobodnym, 73

T

tablice, 33, 217
jednowymiarowe, 34
nieregularne, 41
wielowymiarowe, 36
z haszowaniem, 99
teoria czterech kolorów, 203
tworzenie projektu, 23
typ
boolowski, 16
dynamic, 19
typy danych, 15
referencyjne, 15, 17
wartościowe, 15, 16

U

uruchamianie, 30
usuwanie duplikatów, 121

W

warianty drzew, 123
wartości
całkowite, 16
zmiennoprzecinkowe, 16
wartość logiczna, 16
wejście i wyjście, 25
węzeł, 124, 132, 142
grafu, 178
Wieże Hanoi, 75
wizualizacja drzewa BST, 149
wyliczenia, 17
wyrażenia interpolowane, 18
wyszukiwanie
najkrótszej ścieżki, 207
produktu, 105
wartości, 100

Z

zapisywanie do wyjścia, 27
zbiory, 221
„uporządkowane”, 120
haszowane, 113
złożony ciąg formatujący, 18
znajdowanie drzewa MST, 193, 196
zupełne drzewa binarne, 162

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

C#. Liczy się algorytm i odpowiednia struktura danych!

C# jest nowoczesnym i elastycznym językiem programowania. Aby w pełni skorzystać z jego zalet, trzeba płynnie posługiwać się dostępnymi w nim strukturami danych i algorytmami, pozwalają one bowiem na efektywne organizowanie danych i mają znaczący wpływ na wydajność aplikacji. Z punktu widzenia programisty kluczowe jest ich właściwe zaimplementowanie: wybór właściwej struktury danych i związanego z nią algorytmu stanowi o jakości tworzonego kodu. Na przykład w celu wykonywania wysokowydajnych operacji na zbiorach warto użyć zbioru haszowanego. Inne konstrukcje umożliwiają rozwiązywanie kolejnych problemów.

Dzięki tej książce nauczysz się używania struktur danych i implementacji najważniejszych algorytmów w języku C#. Najpierw zapoznasz się z najprostszymi strukturami danych o swobodnym dostępie — z tablicami oraz listami. Wyjaśniono tu również działanie struktur danych o dostępie sekwencyjnym, opartych na stosach i kolejkach. Przedstawiono zastosowanie słowników, dzięki którym można mapować klucze na wartości i prowadzić szybkie wyszukiwanie. Przystępnie opisano korzystanie z najbardziej zaawansowanych konstrukcji, takich jak drzewo binarne, binarne drzewo poszukiwań, drzewo samorównoważące się i kopiec. W końcowej części książki znajdziesz ciekawą analizę stosowania grafów i związanych z nimi algorytmów, takich jak przeszukiwanie grafu, minimalne drzewo rozpinające, kolorowanie węzłów oraz znajdowanie najkrótszej ścieżki.

Najciekawsze zagadnienia ujęte w książce:

- różne typy danych w C#: wartościowe i referencyjne
- tablice i listy oraz algorytmy sortowania
- operacje na zbiorach oraz wbudowany typ HashSet
- struktury drzewiaste i kopce: binarne, dwumianowe oraz Fibonacciego
- algorytmy oparte na grafach, w tym algorytm Dijkstry

Dr Marcin Jamro jest programistą i architektem różnych aplikacji. Obecnie pełni funkcję prezesa firmy TITUTO Sp. z o.o. z siedzibą w Rzeszowie. Ma również dorobek naukowy, jest autorem kilku publikacji i organizatorem konferencji naukowych. Zdobył certyfikaty: MCP, MCTS i MCPD. Szczególnie interesuje się inżynierią oprogramowania i zarządzaniem projektami.

ISBN 978-83-283-5047-2