

Mizanur Rahman

PHP 7

Algorytmy
i struktury
danych

Helion 

Packt 

Tytuł oryginału: PHP 7 Data Structures and Algorithms

Tłumaczenie: Łukasz Suma

ISBN: 978-83-283-4085-5

Copyright © Packt Publishing 2017

First published in the English language under the title
„PHP 7 Data Structures and Algorithms - (9781786463890)”.

Polish edition copyright © 2018 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 jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/php7al>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	15
<hr/>	
Rozdział 1. Wprowadzenie do algorytmów i struktur danych	19
<hr/>	
Znaczenie algorytmów i struktur danych	20
Znaczenie abstrakcyjnych typów danych (ADT)	23
Różne struktury danych	24
Struktura	25
Tablica	25
Lista jednokierunkowa	26
Lista dwukierunkowa	26
Stos	27
Kolejka	27
Zbiór	28
Mapa (tablica asocjacyjna)	28
Drzewo	28
Graf	29
Serta (kopiec)	29
Rozwiązywanie problemu — podejście algorytmiczne	30
Pisanie pseudokodu	31
Przekształcanie pseudokodu w prawdziwy kod	32
Analiza algorytmu	33
Obliczanie złożoności	34
Zrozumienie notacji dużego O	35
Standardowa biblioteka PHP (SPL) i struktury danych	37
Podsumowanie	38
<hr/>	
Rozdział 2. Zrozumienie tablic PHP	39
<hr/>	
Zrozumienie tablic PHP w lepszy sposób	39
Tablica liczbowa	41
Tablica asocjacyjna	42
Tablica wielowymiarowa	43

Używanie tablicy jako elastycznego sposobu przechowywania danych	44
Używanie wielowymiarowych tablic do reprezentowania struktur danych	45
Tworzenie tablic o stałym rozmiarze za pomocą klasy SplFixedArray	47
Porównanie wydajności zwykłych tablic PHP oraz tablic SplFixedArray	48
Więcej przykładów zastosowania tablicy SplFixedArray	51
Zrozumienie tablic mieszających	53
Implementacja struktury przy użyciu tablicy PHP	54
Implementacja zbioru przy użyciu tablicy PHP	55
Najlepsze zastosowanie tablicy PHP	57
Czy tablica PHP jest zabójcą wydajności?	57
Podsumowanie	58
Rozdział 3. Używanie list	59
Czym jest lista?	59
Różne typy list	63
Lista dwukierunkowa	63
Lista cykliczna	63
Lista wielokierunkowa	64
Wstawianie, usuwanie i wyszukiwanie elementu	64
Wstawianie węzła na pierwszej pozycji	65
Wyszukiwanie węzła	65
Wstawianie przed określonym węzłem	66
Wstawianie po określonym węźle	67
Usuwanie pierwszego węzła	67
Usuwanie ostatniego węzła	68
Wyszukiwanie i usuwanie jednego węzła	69
Odwracanie listy	69
Pobieranie elementu z n-tej pozycji	70
Zrozumienie złożoności list	71
Sprawianie, aby lista była iterowalna	72
Budowanie listy cyklicznej	73
Implementacja listy dwukierunkowej w PHP	75
Operacje na liście dwukierunkowej	75
Wstawianie węzła na pierwszej pozycji	76
Wstawianie węzła na ostatniej pozycji	76
Wstawianie przed określonym węzłem	77
Wstawianie po określonym węźle	78
Usuwanie pierwszego węzła	78
Usuwanie ostatniego węzła	79
Wyszukiwanie i usuwanie jednego węzła	79
Wyświetlanie listy od początku do końca	80
Wyświetlanie listy od końca do początku	80
Złożoność list dwukierunkowych	80
Używanie obiektów klasy PHP SplDoublyLinkedList	81
Podsumowanie	82

Rozdział 4. Konstruowanie stosów i kolejek	83
Zrozumienie stosu	83
Implementacja stosu za pomocą tablicy PHP	84
Zrozumienie złożoności operacji na stosie	87
Implementacja stosu za pomocą listy	88
Używanie klasy SplStack należącej do SPL	90
Rzeczywiste zastosowanie stosu	90
Dopasowywanie zagnieżdżonych nawiasów	91
Zrozumienie kolejki	93
Implementacja kolejki za pomocą tablicy PHP	94
Implementacja kolejki za pomocą listy	95
Używanie klasy SplQueue należącej do SPL	96
Zrozumienie kolejki priorytetowej	96
Sekwencja uporządkowana	97
Sekwencja nieuporządkowana	97
Implementacja kolejki priorytetowej za pomocą listy	97
Implementacja kolejki priorytetowej za pomocą klasy SplPriorityQueue	99
Implementacja kolejki cyklicznej	100
Tworzenie kolejki dwustronnej	102
Podsumowanie	105
Rozdział 5. Stosowanie algorytmów rekurencyjnych	107
Zrozumienie rekurencji	108
Właściwości algorytmów rekurencyjnych	109
Algorytmy rekurencyjne kontra algorytmy iteracyjne	110
Implementacja ciągu Fibonacciego za pomocą rekurencji	111
Implementacja obliczania NWD za pomocą rekurencji	111
Różne rodzaje rekurencji	112
Rekurencja liniowa	112
Rekurencja binarna	112
Rekurencja ogonowa	112
Rekurencja wzajemna	113
Rekurencja zagnieżdżona	113
Budowanie N-poziomowego drzewa kategorii za pomocą rekurencji	114
Budowanie systemu zagnieżdżonych odpowiedzi na komentarze	116
Wyszukiwanie plików i katalogów za pomocą rekurencji	120
Analizowanie algorytmów rekurencyjnych	122
Maksymalna głębokość rekurencji w PHP	123
Używanie rekurencyjnych iteratorów SPL	124
Używanie wbudowanej funkcji PHP array_walk_recursive	125
Podsumowanie	126

Rozdział 6. Zrozumienie i implementacja drzew	127
Definicja i właściwości drzewa	128
Implementacja drzewa przy użyciu języka PHP	130
Różne typy struktur drzewiastych	132
Drzewo binarne	132
Drzewo binarne poszukiwań	133
Samorównoważące się drzewo binarne poszukiwań	133
B-drzewo	135
Drzewo N-arne	135
Zrozumienie drzewa binarnego	135
Implementacja drzewa binarnego	136
Tworzenie drzewa binarnego za pomocą tablicy PHP	138
Zrozumienie binarnego drzewa poszukiwań	140
Wstawianie nowego węzła	141
Wyszukiwanie węzła	141
Wyszukiwanie wartości minimalnej	142
Wyszukiwanie wartości maksymalnej	142
Usuwanie węzła	142
Konstruowanie binarnego drzewa poszukiwań	143
Przechodzenie przez drzewo	151
Przechodzenie bezpośrednio	151
Przechodzenie z wyprzedzeniem	152
Przechodzenie z opóźnieniem	153
Złożoność różnych drzewiastych struktur danych	154
Podsumowanie	155
Rozdział 7. Używanie algorytmów sortowania	157
Zrozumienie sortowania i jego rodzajów	157
Zrozumienie sortowania bąbelkowego	158
Implementacja sortowania bąbelkowego za pomocą języka PHP	159
Złożoność sortowania bąbelkowego	161
Poprawianie algorytmu sortowania bąbelkowego	161
Zrozumienie sortowania przez wybieranie	165
Implementacja sortowania przez wybieranie	167
Złożoność sortowania przez wybieranie	167
Zrozumienie sortowania przez wstawianie	168
Implementacja sortowania przez wstawianie	170
Złożoność sortowania przez wstawianie	171
Zrozumienie technik sortowania wykorzystujących metodę dziel i zwyciężaj	171
Zrozumienie sortowania przez scalanie	171
Implementacja sortowania przez scalanie	173
Złożoność sortowania przez scalanie	174
Zrozumienie sortowania szybkiego	175
Implementacja sortowania szybkiego	176
Złożoność sortowania szybkiego	178
Zrozumienie sortowania kubełkowego	178
Używanie wbudowanych funkcji sortujących PHP	179
Podsumowanie	180

Rozdział 8. Poznawanie technik wyszukiwania	181
Wyszukiwanie liniowe	181
Wyszukiwanie binarne	183
Analiza algorytmu wyszukiwania binarnego	187
Algorytm powtarzalnego wyszukiwania binarnego	187
Przeszukiwanie nieposortowanej tablicy — czy należy ją najpierw posortować?	190
Wyszukiwanie interpolacyjne	191
Wyszukiwanie wykładnicze	192
Wyszukiwanie przy użyciu tablicy mieszającej	193
Wyszukiwanie w drzewach	194
Przeszukiwanie wszerz	194
Przeszukiwanie w głąb	198
Podsumowanie	203
Rozdział 9. Włączanie grafów do akcji	205
Zrozumienie właściwości grafów	205
Wierzchołek	206
Krawędź	206
Sąsiedztwo	207
Incydencja	208
Stopień wchodzący i stopień wychodzący	208
Ścieżka	208
Typy grafów	209
Grafy skierowane	209
Grafy nieskierowane	209
Grafy ważone	210
Skierowane grafy acykliczne	210
Reprezentowanie grafów w PHP	211
Algorytmy BFS i DFS dla grafów	212
Przeszukiwanie wszerz	212
Przeszukiwanie w głąb	214
Sortowanie topologiczne przy użyciu algorytmu Kahna	216
Wyznaczanie najkrótszej ścieżki za pomocą algorytmu Floyda-Warshalla	218
Wyznaczanie najkrótszej ścieżki z pojedynczego źródła za pomocą algorytmu Dijkstry	221
Wyznaczanie najkrótszej ścieżki za pomocą algorytmu Bellmana-Forda	224
Zrozumienie minimalnego drzewa rozpinającego	227
Wyznaczanie minimalnego drzewa rozpinającego za pomocą algorytmu Prima	228
Wyznaczanie minimalnego drzewa rozpinającego za pomocą algorytmu Kruskala	231
Podsumowanie	233

Rozdział 10. Zrozumienie i używanie stert	235
Czym jest sterta?	235
Operacje na stercie	236
Implementacja kopca binarnego w języku PHP	237
Analiza złożoności operacji na stercie	241
Używanie sterty jako kolejki priorytetowej	242
Używanie sortowania przez kopcowanie	245
Używanie klas SplHeap, SplMaxHeap oraz SplMinHeap	248
Podsumowanie	248
Rozdział 11. Rozwiązywanie problemów za pomocą technik zaawansowanych	249
Memoizacja	250
Algorytmy dopasowania do wzorca	252
Implementacja algorytmu Knutha-Morrisa-Pratta	253
Algorytmy zachłanne	255
Implementacja algorytmu kodowania Huffmana	256
Zrozumienie programowania dynamicznego	260
Dyskretny problem plecakowy	261
Znajdowanie długości najdłuższego wspólnego podciągu	262
Sekwencjonowanie DNA przy użyciu programowania dynamicznego	264
Używanie algorytmu z nawrotami do rozwiązywania zagadek	267
System rekomendacji używający wspólnego filtrowania	271
Używanie filtrów Blooma i macierzy rzadkich	274
Podsumowanie	277
Rozdział 12. Obsługa algorytmów i struktur danych przez język PHP	279
Wbudowane w język PHP możliwości związane ze strukturami danych	279
Używanie tablicy PHP	280
Klasy SPL	283
Algorytmy wbudowane	284
Mieszanie	287
Wbudowane możliwości dostępne dzięki PECL	288
Instalacja	289
Interfejsy	290
Wektor	290
Mapa	291
Zbiór	291
Stos i kolejka	293
Kolejka dwustronna	294
Kolejka priorytetowa	294
Podsumowanie	295

Rozdział 13. Funkcyjne struktury danych w języku PHP	297
Zrozumienie programowania funkcyjnego w języku PHP	298
Funkcje pierwszej klasy	299
Funkcje wyższego rzędu	299
Funkcje czyste	299
Funkcje lambda	300
Domknięcia	300
Rozwijanie funkcji	300
Wykonania częściowe	301
Rozpoczęcie pracy z biblioteką Tarsana	302
Implementacja stosu	303
Implementacja kolejki	304
Implementacja drzewa	305
Podsumowanie	306
Skorowidz	307

Konstruowanie stosów i kolejek

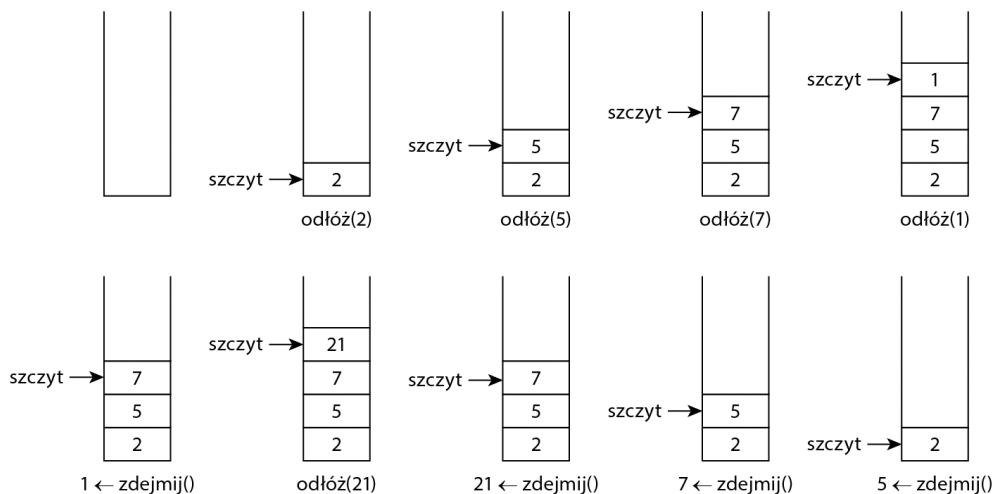
W codziennej praktyce najczęściej używamy dwóch najbardziej popularnych struktur danych. Możemy założyć, że są one wzorowane na obiektach istniejących w prawdziwym świecie, mają jednak ogromny wpływ przede wszystkim na świat komputerowy. Mówimy tu o stosie (ang. *stack*) i kolejce (ang. *queue*). W stosy codziennie układamy nasze książki, pliki dokumentów, talerze i ubrania, podczas gdy z kolejkami mamy do czynienia przy kasach biletowych, na przystankach autobusowych czy na taśmach kasowych supermarketów. Słyszeliśmy też o kolejkach komunikatów w języku PHP będących jedną z najczęściej wykorzystywanych funkcji aplikacji biznesowych. W tym rozdziale przyjrzymy się różnym implementacjom tych popularnych struktur danych, jakimi są stos i kolejka. Poznamy tu zwykle kolejki, kolejki priorytetowe, kolejki cykliczne oraz kolejki dwustronne.

Zrozumienie stosu

Stos jest liniową strukturą danych, która działa zgodnie z regułą **ostatni na wejściu, pierwszy na wyjściu** (ang. *Last-In, First-Out* — **LIFO**). Oznacza to, że stos ma tylko jedną stronę, z której możemy korzystać, aby dodawać i usuwać elementy. Dodawanie elementów na stos określane jest jako odkładanie ich na stos lub umieszczanie na stosie (ang. *push*), a operacja usunięcia nosi nazwę zdejmowania lub pobierania danych ze stosu (ang. *pop*). Jako że do dyspozycji mamy tylko jeden koniec stosu, odkładanie i zdejmowanie realizowane jest zawsze w odniesieniu do tego końca. Element znajdujący się na tym końcu, czyli na samym wierzchu stosu, określany jest mianem szczytu lub wierzchołka (ang. *top*) stosu.

Jeśli przyjrzymy się przedstawionemu poniżej rysunkowi, możemy zauważyć, że każda operacja odłożenia elementu na stos i zdjęcia go ze stosu powoduje, że wierzchołek się zmienia.

Operacje na stosie przeprowadzane są zawsze na jego szczycie, nigdy na początku lub w środku. Musimy zachować szczególną ostrożność przy operacji zdejmowania elementu z pustego stosu oraz odkładania elementu na stos, gdy stos ten jest pełny. Gdy próbujemy umieścić na stosie więcej elementów, niż jest on w stanie przyjąć, możemy otrzymać błąd przepełnienia stosu (ang. *stack overflow error*).



Z wcześniejszego opisu wiemy już, że na stosie można wykonać cztery podstawowe operacje:

- **push:** dodawanie elementu na szczyt stosu;
- **pop:** usunięcie elementu ze szczytu stosu;
- **top:** zwrócenie szczytowego elementu stosu; operacja ta nie jest tożsama z poprzednią, ponieważ w jej przypadku element nie jest usuwany ze stosu, zwracana jest tylko jego wartość;
- **isEmpty:** sprawdzenie, czy stos jest pusty.

Teraz zajmiemy się implementacją stosu w języku PHP, zrobimy to jednak na kilka różnych sposobów. Najpierw spróbujemy zaimplementować stos, korzystając z wbudowanej w PHP tablicy. Następnie przyjrzymy się sposobowi zbudowania stosu bez niej, lecz przy użyciu innych struktur danych, takich jak listy.

Implementacja stosu za pomocą tablicy PHP

Najpierw utworzymy interfejs stosu, z którego będziemy korzystali, tworząc nasze różne implementacje, i który pomoże nam zapewnić, że wszystkie te implementacje właściwie wykonują stawiane przed nimi zadanie. Prosty interfejs stosu wygląda następująco:

```
interface Stack {
    public function push(string $item);
    public function pop();
    public function top();
    public function isEmpty();
}
```

Jak tu widzimy, w interfejsie uwzględniamy wszystkie funkcje stosu, ponieważ implementująca go klasa musi zawierać je wszystkie; w przeciwnym razie w czasie wykonania zgłoszony zostanie błąd krytyczny. Jako że nasz stos implementujemy za pomocą tablicy PHP, opracowując metody odpowiedzialne za operacje odkładania na stos, zdejmowania ze stosu i sprawdzania wartości elementu szczytowego, skorzystamy z pewnych istniejących już funkcji PHP. Stos zdefiniujemy w taki sposób, aby być w stanie określić jego wielkość. Jeśli podjęta zostanie próba zdjęcia elementu z naszego pustego stosu, zgłoszony będzie wyjątek niedoboru (ang. *underflow exception*), a jeśli spróbujemy umieścić na stosie więcej elementów, niż jest on w stanie przyjąć, pojawi się wyjątek przepełnienia (ang. *overflow exception*). Oto kod zapewniający implementację stosu przy użyciu tablicy:

```
class Books implements Stack {
    private $limit;
    private $stack;

    public function __construct(int $limit = 20) {
        $this->limit = $limit;
        $this->stack = [];
    }

    public function pop(): string {
        if ($this->isEmpty()) {
            throw new UnderflowException('Stos jest pusty');
        } else {
            return array_pop($this->stack);
        }
    }

    public function push(string $newItem) {
        if (count($this->stack) < $this->limit) {
            array_push($this->stack, $newItem);
        } else {
            throw new OverflowException('Stos jest pełny');
        }
    }
}
```

```

    public function top(): string {
        return end($this->stack);
    }

    public function isEmpty(): bool {
        return empty($this->stack);
    }
}

```

Przeanalizujemy teraz nasz kod implementujący stos. Klasie stosu nadaliśmy nazwę Books, ale moglibyśmy nazwać ją dowolnie inaczej, pod warunkiem że ta nazwa spełniałaby formalne wymogi języka. Implementację zaczynamy od metody `__construct`, która umożliwia nam określenie maksymalnej liczby elementów przechowywanych na stosie. Domyślną wartością jest tu 20. Następna metoda stanowi implementację operacji zdejmowania elementu ze stosu:

```

    public function pop(): string {
        if ($this->isEmpty()) {
            throw new UnderflowException('Stos jest pusty');
        } else {
            return array_pop($this->stack);
        }
    }
}

```

Metoda `pop` zwraca łańcuch znakowy, jeśli stos nie jest pusty. Sprawdzenie tego warunku odbywa się za pomocą specjalnej metody, którą zdefiniowaliśmy w klasie stosu. Jeśli stos jest pusty, zgłoszony zostaje wyjątek `UnderFlowException`, zdefiniowany w SPL. Jeśli nie ma elementu do zdjęcia, możemy w ten sposób sprawić, aby ta operacja nie była wykonywana. Jeśli stos nie jest pusty, korzystamy z zapewnianej przez PHP funkcji `array_pop`, aby zwrócić ostatni element naszej tablicy.

W metodzie `push` odbywa się działanie odwrotne do tego, które wykonuje metoda `pop`. Najpierw sprawdzamy, czy stos jest pełny. Jeśli nie, dodajemy do jego końca element będący łańcuchem znakowym, korzystając w tym celu z zapewnianej przez PHP funkcji `array_push`. Jeśli stos jest pełny, zgłaszamy wyjątek `OverflowException`, który został zdefiniowany w SPL. Metoda `top` zwraca element znajdujący się na szczycie stosu. Metoda `isEmpty` umożliwia sprawdzenie, czy stos jest pusty.

Ponieważ korzystamy z języka PHP 7, używamy tu zarówno deklaracji typów skalarnych na poziomie metod, jak i typów wartości zwracanych przez metody.

Aby skorzystać z naszej zaimplementowanej właśnie klasy, musimy pomyśleć o przykładzie, w którym moglibyśmy użyć wszystkich zdefiniowanych dla niej operacji. Napişmy niewielki program odwzorowujący stos książek. Oto jego kod:

```

try {
    $programmingBooks = new Books(10);
    $programmingBooks->push("Wprowadzenie do PHP7");
}

```

```

$programmingBooks->push("Mistrzowski JavaScript");
$programmingBooks->push("Samouczek MySQL Workbench");
echo $programmingBooks->pop()."\n";
echo $programmingBooks->top()."\n";
} catch (Exception $e) {
    echo $e->getMessage();
}

```

Utworzyliśmy tu instancję stosu książek, w której będziemy przechowywać tytuły naszych książek programistycznych. Wykonaliśmy też trzy operacje odłożenia elementu na stos. Ostatnią umieszczoną na nim książką był "Samouczek MySQL Workbench". Gdy po tych trzech operacjach umieszczenia elementu na stosie wykonaliśmy operację zdjęcia z niego elementu, zwrócony został ten właśnie tytuł. Następujące po tym wywołanie metody `top` zwróciło tytuł "Mistrzowski JavaScript", który był w tym momencie szczytowym elementem stosu. Cały kod umieściliśmy w bloku `try...catch`, dzięki czemu możemy obsługiwać wyjątek, który może zostać zgłoszony w momencie wystąpienia przepełnienia lub niedoboru. Wykonanie przedstawionego powyżej fragmentu kodu powoduje wyświetlenie na ekranie następujących danych wyjściowych:

```

Samouczek MySQL Workbench
Mistrzowski JavaScript

```

Przyjrzyjmy się teraz złożoności różnych operacji na stosie, który przed chwilą zaimplementowaliśmy.

Zrozumienie złożoności operacji na stosie

Poniżej zostały zebrane złożoności czasowe różnych operacji na stosie. W najgorszym przypadku są one następujące:

Operacja	Złożoność czasowa
<code>pop</code>	$O(1)$
<code>push</code>	$O(1)$
<code>top</code>	$O(1)$
<code>isEmpty</code>	$O(1)$

Jako że w przypadku stosu operujemy zawsze tylko na jednym końcu struktury, jeśli chcemy wyszukać element w stosie, musimy przeszukać całą tworzącą go listę. To samo dotyczy dostępu do określonego elementu należącego do stosu. Wykonywanie tego typu operacji jest dobrą praktyką, lecz jeśli już koniecznie chcemy je przeprowadzać, musimy pamiętać, że ich złożoność czasowa bierze się nie tylko z ogólnych operacji na stosie.

Operacja	Złożoność czasowa
dostęp	$O(n)$
wyszukiwanie	$O(n)$

Złożoność pamięciowa stosu wynosi zawsze $O(n)$.

Jak dotąd dowiedzieliśmy się, jak można zaimplementować stos, używając tablicy PHP oraz wbudowanych w język funkcji `array_pop` i `array_push`. Moglibyśmy jednak zignorować fakt istnienia tych funkcji i zaimplementować stos, korzystając z operacji wykonywanych na tablicy ręcznie, moglibyśmy też użyć wbudowanych funkcji `array_shift` oraz `array_unshift`.

Implementacja stosu za pomocą listy

W rozdziale 3. pt. „Używanie list” dowiedzieliśmy się, jak tworzyć listy. Przekonaliśmy się, że korzystając z listy, możemy wstawiać węzły na jej końcu, usuwać je z tego końca, wstawiać w środku listy i na jej początku itd. Jeśli weźmiemy pod uwagę jedynie operacje wstawiania elementów na końcu listy oraz usuwanie ich z jej końca, będziemy mieli do czynienia ze strukturą danych przypominającą stos. Użyjmy zatem opracowanej w poprzednim rozdziale klasy `LinkedList`, aby zaimplementować stos. Oto odpowiedni kod:

```
class BookList implements Stack {

    private $stack;

    public function __construct() {
        $this->stack = new LinkedList();
    }

    public function pop(): string {

        if ($this->isEmpty()) {
            throw new UnderflowException('Stos jest pusty');
        } else {
            $lastItem = $this->top();
            $this->stack->deleteLast();
            return $lastItem;
        }
    }

    public function push(string $newItem) {
        $this->stack->insert($newItem);
    }
}
```



```

public function top(): string {
    return $this->stack->getNthNode($this->stack->getSize())->data;
}

public function isEmpty(): bool {
    return $this->stack->getSize() == 0;
}
}

```

Przeanalizujmy każdy blok kodu tworzącego naszą nową klasę, aby zrozumieć, co się w niej dzieje. Jeśli zaczniemy od początku, zauważymy, że w metodzie `__construct` tworzony jest nowy obiekt klasy `LinkedList` i że jest on (zamiast tablicy, jak to było w naszym poprzednim przykładzie) przypisywany do właściwości `$stack`. Zakładamy tu, że klasa `LinkedList` zostaje załadowana automatycznie lub też odpowiedni plik jest włączony do skryptu. Skupmy się teraz na operacji odkładania elementu na stos, która jest naprawdę prosta. Musimy w jej przypadku po prostu wstawić nowy węzeł na listę. Jako że nie istnieją żadne ograniczenia dotyczące długości listy, nie sprawdzamy tu przepełnienia.

W naszej implementacji listy nie było metody odpowiedzialnej za zwracanie ostatniego węzła. Mieliśmy możliwość wstawienia ostatniego elementu i usunięcia poprzedniego ostatniego elementu, tutaj jednak potrzebna nam jest metoda, która zwróci nam ostatni węzeł bez usuwania go z listy. Aby ją opracować, zapewniając tym samym mechanizm działania metody `top` dla naszego stosu, możemy skorzystać z metod `getNthNode` oraz `getSize` należących do implementacji klasy `LinkedList`. Możemy w ten sposób pobrać węzeł. Musimy tu jednak pamiętać o tym, że chodzi nam o wartość węzła będącą łańcuchem znakowym, nie zaś o cały obiekt węzła. To właśnie z tego powodu zwracamy wartość odpowiedniej właściwości otrzymanego węzła.

Podobnie jak metoda `top`, metoda `pop` również musi zwrócić dane ostatniego węzła, ta ostatnia ma go jednak jeszcze usunąć z listy. Aby wykonać te operacje, korzystamy z metody `top`, a następnie z metody `deleteLast` należącej do klasy `LinkedList`. Teraz uruchommy przykładowy kod, aby sprawdzić działanie zaimplementowanej przed chwilą klasy `BookList`, która ma spełniać funkcję stosu. Oto odpowiedni kod:

```

try {
    $programmingBooks = new BookList();
    $programmingBooks->push("Wprowadzenie do PHP7");
    $programmingBooks->push("Mistrzowski JavaScript");
    $programmingBooks->push("Samouczek MySQL Workbench");
    echo $programmingBooks->pop()."\n";
    echo $programmingBooks->pop()."\n";
    echo $programmingBooks->top()."\n";
} catch (Exception $e) {
    echo $e->getMessage();
}

```

Ten kod wygląda bardzo podobnie do poprzedniego przykładu, który niedawno uruchomiliśmy, tutaj jednak spróbowałeś wykonać dwie operacje zdjęcia elementu ze stosu oraz jedną operację odczytania danych elementu szczytowego. Dane wyjściowe mają zatem następującą postać:

```
Samouczek MySQL Workbench
Mistrzowski JavaScript
Wprowadzenie do PHP7
```

Jeśli znamy podstawową zasadę działania stosu oraz sposób, w jaki można ją zaimplementować w praktyce, do utworzenia stosu możemy użyć tablicy, listy jednokierunkowej lub listy dwukierunkowej. Jako że mieliśmy się już okazję poznać implementację stosu za pomocą tablicy i listy jednokierunkowej, przyjrzyjmy się teraz implementacji tej struktury zapewnianej przez SPL, w przypadku której wykorzystywana jest tak naprawdę lista dwukierunkowa.

Używanie klasy SplStack należącej do SPL

Jeśli nie chcemy tworzyć od podstaw własnej wersji stosu, możemy skorzystać z jego implementacji zapewnianej przez SPL. Używa się jej bardzo łatwo i wymaga ona napisania tylko niewielkiej ilości kodu. Jak już wiemy, klasa SplStack korzysta z klasy SplDoublyLinkedList. Oferuje wszelkie niezbędne operacje, takie jak odkładanie elementu na stos, zdejmowanie go, przesuwanie do przodu oraz do tyłu itd. Aby opracować przykład podobny do przedstawionego wcześniej, musimy tylko napisać następujące wiersze kodu:

```
$books = new SplStack();
$books->push("Wprowadzenie do PHP7");
$books->push("Mistrzowski JavaScript");
$books->push("Samouczek MySQL Workbench");
echo $books->pop() . "\n";
echo $books->top() . "\n";
```

To prawda, że stos najprościej jest zbudować, korzystając z klasy SplStack. Możemy jednak równie dobrze zaimplementować go samodzielnie, używając tablicy PHP lub listy, a wybór rozwiązania zależy wyłącznie od nas.

Rzeczywiste zastosowanie stosu

Stos ma wiele zastosowań w nowoczesnych aplikacjach i jest używany niemal wszędzie, czego przykładami mogą być historia odwiedzanych stron przechowywana przez przeglądarkę internetową oraz powszechnie wykorzystywany w środowisku programistów ślad stosu. Korzystając ze stosu, spróbujemy teraz rozwiązać pewien realny problem.

Dopasowywanie zagnieżdżonych nawiasów

Pierwszą czynnością, którą musimy wykonać przy rozwiązywaniu równań lub obliczaniu wartości wyrażeń matematycznych, jest sprawdzenie poprawności zagnieżdżonych nawiasów. Jeśli nie są one zagnieżdżone prawidłowo, wówczas obliczeń może nie dać się wykonać lub ich wynik może być nieprawidłowy. Przyjrzyjmy się kilku przykładom:

$$8 * (9 - 2) + \{ (4 * 5) / (2 * 2) \}$$

$$5 * 8 * 9 / (3 * 2)$$

$$[\{ (2 * 7) + (15 - 3) \}]$$

Z przedstawionych tu wyrażeń tylko pierwsze jest poprawne; pozostałe dwa są nieprawidłowe, ponieważ nawiasy nie są w nich zagnieżdżone prawidłowo. Aby stwierdzić, czy nawiasy są poprawnie zagnieżdżone, możemy opracować rozwiązanie wykorzystujące stos. Oto pseudokod opisujący algorytm takiego rozwiązania:

```

valid = true
s = empty stack
for (each character of the string) {
    if(character = ( or { or [ )
        s.push(character)
    else if (character = ) or } or ] ) {
        if(s is empty)
            valid = false
        last = s.pop()
        if(last is not opening parentheses of character)
            valid = false
    }
}
if(s is not empty)
    valid = false

```

Jeśli przyjrzymy się temu pseudokodowi, okaże się on dość nieskomplikowany. Jego działanie polega na tym, aby ignorować wszelkie liczby, operandy i puste znaki występujące w łańcuchu tekstowym i brać pod uwagę wyłącznie nawiasy okrągłe, klamrowe oraz kwadratowe. Jeśli w łańcuchu pojawiają się jakieś nawiasy otwierające, umieszczamy je na stosie. Jeśli występują tam nawiasy zamykające, zdejmujemy elementy ze stosu. Jeśli nawias zdjęty ze stosu nie jest nawiasem otwierającym, który pasuje do znalezionego nawiasu zamykającego, wówczas wyrażenie jest niepoprawne. Jeśli łańcuch jest prawidłowy, na końcu pętli stos powinien być pusty. Jeśli nie jest, to mamy jakieś nadmiarowe nawiasy, a zatem wyrażenie jest nieprawidłowe. Przekształćmy teraz ten algorytm w program:

```

function expressionChecker(string $expression): bool {
    $valid = TRUE;
    $stack = new SplStack();

    for ($i = 0; $i < strlen($expression); $i++) {
        $char = substr($expression, $i, 1);

        switch ($char) {

```

```

        case '(':
        case '{':
        case '[':
            $stack->push($char);
            break;
        case ')':
        case '}':
        case ']':
            if ($stack->isEmpty()) {
                $valid = FALSE;
            } else {
                $last = $stack->pop();
                if (($char == "(" && $last != ")" || ($char == "{" && $last != "{")
                    || ($char == "]" && $last != "[")) {
                    $valid = FALSE;
                }
            }
            break;
        }
        if (!$valid)
            break;
    }
    if (!$stack->isEmpty()) {
        $valid = FALSE;
    }

    return $valid;
}

```

Uruchommy teraz tę funkcję, podając jej trzy przedstawione wcześniej wyrażenia:

```

$expressions = [];
$expressions[] = "8 * (9 -2) + { (4 * 5) / ( 2 * 2) }";
$expressions[] = "5 * 8 * 9 / ( 3 * 2 )";
$expressions[] = "[{ (2 * 7) + ( 15 - 3) ]";

foreach ($expressions as $expression) {
    $valid = expressionChecker($expression);

    if ($valid) {
        echo "Wyrażenie jest prawidłowe \n";
    } else {
        echo "Wyrażenie jest nieprawidłowe \n";
    }
}

```

Wykonanie tego fragmentu kodu spowoduje wyświetlenie na ekranie następujących danych wyjściowych, które są w stu procentach zgodne z naszymi oczekiwaniami:

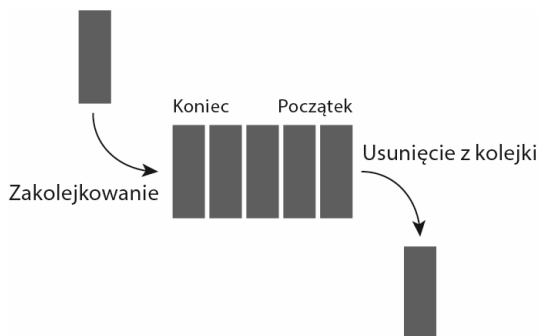
```

Wyrażenie jest prawidłowe
Wyrażenie jest nieprawidłowe
Wyrażenie jest nieprawidłowe

```

Zrozumienie kolejki

Kolejka to następna specjalna, liniowa struktura danych, działająca jednak zgodnie z zasadą **pierwszy na wejściu, pierwszy na wyjściu** (ang. *First-In, First-Out* — **FIFO**). Operacje odbywają się na dwóch końcach kolejki: jeden z nich służy do dodawania elementów, a drugi do usuwania. Odróżnia to kolejkę od stosu, w przypadku którego obydwa te działania odbywały się tylko na jednym końcu. Wstawianie elementów do kolejki odbywa się zawsze z tyłu lub na końcu kolejki. Usuwanie elementów przeprowadza się na jej początku czy też z przodu. Operacja dodawania nowego elementu do kolejki znana jest jako *zakolejkowanie* (ang. *enqueue*), a operację usuwania można określić słowem „wykolejkowanie”, „zdekolejkowanie” lub po prostu jako obsługę elementu (ang. *dequeue*). Pobieranie elementu znajdującego się na początku kolejki bez usuwania go znane jest jako *zerkanie* lub *podglądanie* (ang. *peek*) i stanowi operację analogiczną do operacji wykonywanej na stosie przez metodę *top*. Sposób działania kolejki został przedstawiony na rysunku poniżej.



Interfejs kolejki powinien mieć następującą definicję:

```
interface Queue {
    public function enqueue(string $item);

    public function dequeue();

    public function peek();

    public function isEmpty();
}
```

Podobnie jak to było w przypadku stosu, kolejkę możemy zaimplementować na różne sposoby. Najpierw zrealizujemy ją za pomocą tablicy PHP, następnie przy użyciu klasy `LinkedList`, a na koniec — korzystając z klasy `SplQueue`.

Implementacja kolejki za pomocą tablicy PHP

Teraz zajmiemy się implementacją kolejki przy użyciu tablicy PHP. Wiemy już, że możemy zastosować funkcję `array_push` w celu dodania elementu na końcu tablicy. Aby usunąć jej pierwszy element, można skorzystać z funkcji `array_shift` zapewnianej przez PHP. Pierwszy element kolejki da się podejrzeć za pomocą funkcji `current`. W związku z powyższym kod naszej kolejki może wyglądać następująco:

```
class AgentQueue implements Queue {

    private $limit;
    private $queue;

    public function __construct(int $limit = 20) {
        $this->limit = $limit;
        $this->queue = [];
    }

    public function dequeue(): string {
        if ($this->isEmpty()) {
            throw new UnderflowException('Kolejka jest pusta');
        } else {
            return array_shift($this->queue);
        }
    }

    public function enqueue(string $newItem) {
        if (count($this->queue) < $this->limit) {
            array_push($this->queue, $newItem);
        } else {
            throw new OverflowException('Kolejka jest pełna');
        }
    }

    public function peek(): string {
        return current($this->queue);
    }

    public function isEmpty(): bool {
        return empty($this->queue);
    }
}
```

Kierujemy się tu tą samą regułą, zgodnie z którą postępowaliśmy, tworząc nasz stos. Definiujemy zatem kolejkę o stałej wielkości (czy też długości), sprawdzając, czy nie mamy do czynienia z przepełnieniem lub niedoborem. W celu sprawdzenia naszej implementacji kolejki utworzymy kolejkę agentów telefonicznego centrum obsługi. Oto kod, w którym przeprowadzane są poszczególne operacje na kolejce:

```

try {
    $agents = new AgentQueue(10);
    $agents->enqueue("Franek");
    $agents->enqueue("Janek");
    $agents->enqueue("Krzysiek");
    $agents->enqueue("Adrian");
    $agents->enqueue("Michał");
    echo $agents->dequeue()."\n";
    echo $agents->dequeue()."\n";
    echo $agents->peek()."\n";
} catch (Exception $e) {
    echo $e->getMessage();
}

```

W wyniku wykonania tego kodu na ekranie pojawiają się następujące dane wyjściowe:

```

Franek
Janek
Krzysiek

```

Implementacja kolejki za pomocą listy

Tak jak to było w przypadku stosu, również tutaj do opracowania kolejki zamierzamy użyć naszej implementacji listy zaprezentowanej w rozdziale 3. pt. „Używanie list”. Możemy tu użyć metody `insert` do wstawiania elementów zawsze na końcu kolejki, metody `deleteFirst`, aby obsłużyć operację usuwania elementów z kolejki, zaś metody `getNthNode` do podglądania pierwszego elementu. Oto przykładowa implementacja kolejki przy użyciu listy jednokierunkowej:

```

class AgentQueue implements Queue {

    private $limit;
    private $queue;

    public function __construct(int $limit = 20) {
        $this->limit = $limit;
        $this->queue = new LinkedList();
    }

    public function dequeue(): string {
        if ($this->isEmpty()) {
            throw new UnderflowException('Kolejka jest pusta');
        } else {
            $lastItem = $this->peek();
            $this->queue->deleteFirst();
            return $lastItem;
        }
    }
}

```

```

public function enqueue(string $newItem) {
    if ($this->queue->getSize() < $this->limit) {
        $this->queue->insert($newItem);
    } else {
        throw new OverflowException('Kolejka jest pełna');
    }
}

public function peek(): string {
    return $this->queue->getNthNode(1)->data;
}

public function isEmpty(): bool {
    return $this->queue->getSize() == 0;
}
}

```

Używanie klasy SplQueue należącej do SPL

Jeśli nie chcemy męczyć się implementacją funkcji kolejki i nie mamy problemu z zastosowaniem rozwiązania oferowanego przez bibliotekę standardową, możemy użyć klasy SplQueue do zaspokojenia naszych podstawowych potrzeb związanych z tego rodzaju strukturą danych. Musimy tylko pamiętać o jednym: klasa SplQueue nie zapewnia funkcji podglądania wartości pierwszego elementu kolejki. Z tego powodu powinniśmy skorzystać z funkcji bottom w celu uzyskania pierwszego elementu kolejki. Oto prosta wykorzystująca obiekt klasy SplQueue implementacja kolejki reprezentującej przedstawiony wcześniej przykład AgentQueue:

```

$agents = new SplQueue();
$agents->enqueue("Franek");
$agents->enqueue("Janek");
$agents->enqueue("Krzysiek");
$agents->enqueue("Adrian");
$agents->enqueue("Michał");
echo $agents->dequeue()."\n";
echo $agents->dequeue()."\n";
echo $agents->bottom()."\n";

```

Zrozumienie kolejki priorytetowej

Kolejka priorytetowa (ang. *priority queue*) to specjalny rodzaj kolejki, w przypadku której elementy są wstawiane i usuwane zgodnie z priorytetem. W świecie programowania komputerowego zastosowanie kolejki priorytetowej jest ogromne. Powiedzmy, że mamy bardzo duży system kolejkowania wiadomości poczty elektronicznej, który wykorzystujemy do rozsyłania

comiesięcznego biuletynu. Co gdy zachodzi potrzeba rozesłania do użytkowników jakiejś niezwykle pilnej wiadomości za pomocą tego systemu? Jako że zgodnie z ogólną zasadą działania kolejek każdy element dodaje się na jej końcu, doręczenie naszej wiadomości będzie bardzo mocno opóźnione. Aby rozwiązać ten problem, możemy skorzystać z kolejki priorytetowej. W takim przypadku do każdego węzła przypisuje się pewien priorytet i wszystkie węzły sortuje się zgodnie z ich priorytetami. Element o wyższym priorytecie zostanie przeniesiony na początek listy, w związku z czym zostanie on obsłużony wcześniej niż węzły o niższych priorytetach.

Kolejkę priorytetową można zbudować na dwa sposoby.

Sekwencja uporządkowana

Jeśli implementując kolejkę priorytetową, zdecydowaliśmy się użyć sekwencji uporządkowanej (ang. *ordered sequence*), możemy w jej przypadku zastosować porządek rosnący lub malejący. Dobrą stroną korzystania z tego rodzaju sekwencji jest to, że możemy szybko w niej znaleźć lub z niej usunąć element o najwyższym priorytecie; złożoność tego rodzaju operacji wynosi zaledwie $O(1)$. Więcej czasu zajmie jednak wstawianie elementu, ponieważ będzie ono wymagało sprawdzenia każdego elementu kolejki w celu umieszczenia nowego węzła w miejscu odpowiednim ze względu na jego priorytet.

Sekwencja nieuporządkowana

Zastosowanie sekwencji nieuporządkowanej (ang. *unordered sequence*) nie zmusza nas do przechodzenia przez każdy element kolejki w celu umieszczenia w niej nowo dodanego elementu. Dodaje się go do końca kolejki, zgodnie z ogólną zasadą działania kolejek. Dzięki temu złożoność operacji kolejki wynosi $O(1)$. Jeśli jednak chcemy wyszukać lub usunąć element o najwyższym priorytecie, musimy przejść przez każdy element kolejki, aby znaleźć właściwy węzeł. Co za tym idzie, rozwiązanie to nie jest najlepsze, gdy chodzi o operacje wyszukiwania.

Teraz zajmiemy się pisaniem kodu implementującego kolejkę priorytetową przy użyciu uporządkowanej sekwencji realizowanej za pomocą listy.

Implementacja kolejki priorytetowej za pomocą listy

Jak dotąd listy, z którymi mieliśmy do czynienia, w każdym swoim węzle przechowywały tylko jedną wartość stanowiącą dane węzła. Teraz zachodzi potrzeba zapisania innej wartości, która określa priorytet. Aby to osiągnąć, musimy zmienić implementację naszej klasy `ListNode` w następujący sposób:

```

class ListNode {

    public $data = NULL;
    public $next = NULL;
    public $priority = NULL;

    public function __construct(string $data = NULL, int $priority = NULL) {
        $this->data = $data;
        $this->priority = $priority;
    }

}

```

Dzięki temu zarówno dane, jak i priorytet stanowią teraz elementy składowe naszego węzła. Aby uwzględnić priorytet podczas wstawiania nowych węzłów, musimy również zmienić implementację metody insert należącej do klasy LinkedList. Oto zmodyfikowany kod:

```

public function insert(string $data = NULL, int $priority = NULL) {
    $newNode = new ListNode($data, $priority);
    $this->_totalNode++;

    if ($this->_firstNode === NULL) {
        $this->_firstNode = &$newNode;
    } else {
        $previous = $this->_firstNode;
        $currentNode = $this->_firstNode;
        while ($currentNode !== NULL) {
            if ($currentNode->priority < $priority) {

                if ($currentNode == $this->_firstNode) {
                    $previous = $this->_firstNode;
                    $this->_firstNode = $newNode;
                    $newNode->next = $previous;
                    return;
                }
                $newNode->next = $currentNode;
                $previous->next = $newNode;
                return;
            }
            $previous = $currentNode;
            $currentNode = $currentNode->next;
        }
    }
    return TRUE;
}

```

Jak widać, nasza metoda insert została zmieniona w taki sposób, aby przyjmować w roli argumentów zarówno dane, jak i priorytet, a także aby uwzględnić je w czasie realizowania operacji. Jak zwykle pierwszym działaniem jest tu utworzenie nowego węzła i zinkrementowanie licznika węzłów. Przy wstawianiu istnieją trzy możliwe sytuacje:

- Lista jest pusta, dlatego nowy węzeł staje się pierwszym węzłem listy.
- Lista nie jest pusta, ale nowy element ma najwyższy priorytet, dlatego staje się pierwszym węzłem, a element, który wcześniej nim był, znajduje się po nowo wstawionym.
- Lista nie jest pusta, a nowy element nie ma najwyższego priorytetu, dlatego zostaje wstawiony gdzieś w środku listy, a być może na jej końcu.

Tworząc naszą implementację, uwzględniliśmy wszystkie te trzy przypadki. Dzięki temu element o najwyższym priorytecie mamy zawsze na początku kolejki. Uruchommy teraz przykład z kolejką `AgentQueue` zbudowaną na bazie naszego nowego kodu, tak jak zostało to pokazane poniżej.

```
try {
    $agents = new AgentQueue(10);
    $agents->enqueue("Franek", 1);
    $agents->enqueue("Janek", 2);
    $agents->enqueue("Krzysiek", 3);
    $agents->enqueue("Adrian", 4);
    $agents->enqueue("Michał", 2);
    $agents->display();
    echo $agents->dequeue()."\n";
    echo $agents->dequeue()."\n";
} catch (Exception $e) {
    echo $e->getMessage();
}
```

Gdybyśmy nie korzystali z priorytetów, wówczas nasza kolejka powinna zawierać elementy ułożone w następującym porządku: Franek, Janek, Krzysiek, Adrian i Michał. Ponieważ jednak dodaliśmy priorytety, dane wyjściowe wyglądają jak niżej.

```
Adrian
Krzysiek
Janek
Michał
Franek
```

Jako że element Adrian ma najwyższy priorytet, jest umieszczony na początku kolejki, choć został do niej wstawiony jako czwarty.

Implementacja kolejki priorytetowej za pomocą klasy `SplPriorityQueue`

Język PHP zapewnia wsparcie dla implementacji kolejki priorytetowej za pomocą SPL. Do utworzenia tego rodzaju struktury danych możemy wykorzystać klasę `SplPriorityQueue`. Oto ta sama kolejka, którą implementowaliśmy w poprzednim przykładzie za pomocą listy, zrealizowana jednak tym razem przy użyciu rozwiązania zapewnianego przez SPL:

```

class MyPQ extends SplPriorityQueue {

    public function compare($priority1, $priority2) {
        return $priority1 <=> $priority2;
    }

}

$agents = new MyPQ();

$agents->insert("Franek", 1);
$agents->insert("Janek", 2);
$agents->insert("Krzysiek", 3);
$agents->insert("Adrian", 4);
$agents->insert("Michał", 2);

// Tryb ekstrakcji
$agents->setExtractFlags(MyPQ::EXTR_BOTH);

// Przejście do SZCZYTU
$agents->top();

while ($agents->valid()) {
    $current = $agents->current();
    echo $current['dane'] . "\n";
    $agents->next();
}

```

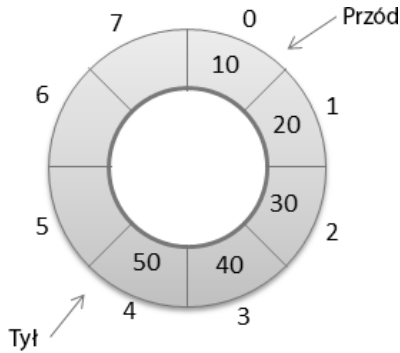
Przedstawiony powyżej kod wygeneruje te same dane wyjściowe co przykład wykorzystujący listę. Dodatkową przewagą rozwiązania polegającego na użyciu klasy MyPQ rozszerzającej klasę SplPriorityQueue jest to, że możemy tu wybrać sposób sortowania elementów (tj. to, czy mają być one ułożone w kolejności rosnącej, czy też malejącej). W tym przykładzie wybraliśmy porządek malejący, korzystając z dostępnego w PHP operatora połączonego porównania, zwanego ze względu na swój wygląd operatorem statku kosmicznego (ang. *spaceship operator*).

Kolejki priorytetowe są zwykle implementowane za pomocą sterty. Gdy przejdziemy do rozdziału poświęconego sterce, zajmiemy się również implementacją kolejki priorytetowej przy użyciu tej struktury danych.

Implementacja kolejki cyklicznej

Korzystając ze standardowej kolejki, za każdym razem gdy usuwamy z niej element, musimy odpowiednio przesunąć całą kolejkę. Aby rozwiązać ten problem, możemy skorzystać z kolejki cyklicznej (ang. *circular queue*), w której po końcu następuje początek, dzięki czemu powstaje koło czy też cykl. Ten szczególny typ kolejki wymaga specjalnych obliczeń wykonywanych przy operacjach dodawania elementu do kolejki i usuwania elementu z kolejki, a związanych

z jej końcem, początkiem oraz wielkością. Kolejki cykliczne mają zawsze stały rozmiar i znane są także jako bufony cykliczne (ang. *circular buffers*) lub bufony pierścieniowe (ang. *ring buffers*). Na zamieszczonym poniżej rysunku przedstawiony został schemat działania kolejki cyklicznej.



Kolejkę cykliczną można zaimplementować za pomocą tablicy PHP. Może ona być w tym celu wydajnie wykorzystana, ponieważ musimy tu obliczyć pozycje końca i początku kolejki. Oto przykład kolejki cyklicznej:

```
class CircularQueue implements Queue {

    private $queue;
    private $limit;
    private $front = 0;
    private $rear = 0;

    public function __construct(int $limit = 5) {
        $this->limit = $limit;
        $this->queue = [];
    }

    public function size() {
        if ($this->rear > $this->front)
            return $this->rear - $this->front;
        return $this->limit - $this->front + $this->rear;
    }

    public function isEmpty() {
        return $this->rear == $this->front;
    }

    public function isFull() {
        $diff = $this->rear - $this->front;
        if ($diff == -1 || $diff == ($this->limit - 1))
            return true;
        return false;
    }
}
```

```

public function enqueue(string $item) {
    if ($this->isFull()) {
        throw new OverflowException("Kolejka jest pełna.");
    } else {
        $this->queue[$this->rear] = $item;
        $this->rear = ($this->rear + 1) % $this->limit;
    }
}

public function dequeue() {
    $item = "";
    if ($this->isEmpty()) {
        throw new UnderflowException("Kolejka jest pusta");
    } else {
        $item = $this->queue[$this->front];
        $this->queue[$this->front] = NULL;
        $this->front = ($this->front + 1) % $this->limit;
    }
    return $item;
}

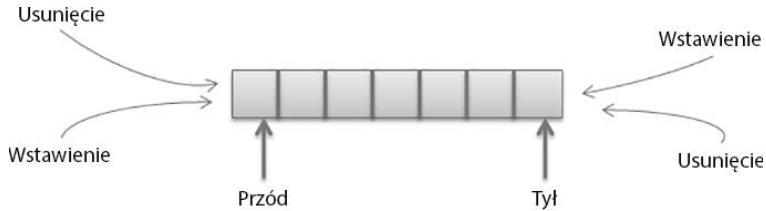
public function peek() {
    return $this->queue[$this->front];
}
}

```

Ponieważ uznajemy 0 za znacznik początku kolejki, jej całkowity rozmiar będzie wynosił `limit - 1`.

Tworzenie kolejki dwustronnej

Do tej pory implementowaliśmy kolejki, których jedna strona, określana jako koniec lub tył (ang. *rear*), używana była do dodawania elementów, zaś druga, znana jako początek lub przód (ang. *front*), umożliwiała usuwanie elementów. Ogólnie rzecz biorąc, każda strona kolejki powinna być zatem wykorzystywana w konkretnym celu. Co jednak, gdybyśmy chcieli dodawać i usuwać elementy z obu stron kolejki? Da się to zrobić za pomocą tzw. kolejki dwustronnej, określanej też jako kolejka podwójna (ang. *double-ended queue*, *deque*). W kolejce takiej obydwie strony mogą być używane do operacji dodawania i usuwania węzłów. Jeśli spojrzymy na naszą implementację kolejki wykorzystującą listę, przekonamy się, że można w jej przypadku wstawiać element na pierwszym i na ostatnim miejscu oraz usuwać element z pierwszego i ostatniego miejsca. Tworząc nową klasę kolejki dwustronnej z wykorzystaniem tych możliwości, możemy łatwo osiągnąć cel, o który nam chodzi. Sposób działania tego rodzaju kolejki został pokazany poniżej.



Oto implementacja kolejki dwustronnej:

```
class DeQueue {

    private $limit;
    private $queue;

    public function __construct(int $limit = 20) {
        $this->limit = $limit;
        $this->queue = new LinkedList();
    }

    public function dequeueFromFront(): string {

        if ($this->isEmpty()) {
            throw new UnderflowException('Kolejka jest pusta');
        } else {
            $lastItem = $this->peekFront();
            $this->queue->deleteFirst();
            return $lastItem;
        }
    }

    public function dequeueFromBack(): string {

        if ($this->isEmpty()) {
            throw new UnderflowException('Kolejka jest pusta');
        } else {
            $lastItem = $this->peekBack();
            $this->queue->deleteLast();
            return $lastItem;
        }
    }

    public function enqueueAtBack(string $newItem) {
        if ($this->queue->getSize() < $this->limit) {
            $this->queue->insert($newItem);
        } else {
            throw new OverflowException('Kolejka jest pełna');
        }
    }

    public function enqueueAtFront(string $newItem) {
        if ($this->queue->getSize() < $this->limit) {
```

```

        $this->queue->insertAtFirst($NewItem);
    } else {
        throw new OverflowException('Kolejka jest pełna');
    }
}

public function peekFront(): string {
    return $this->queue->getNthNode(1)->data;
}

public function peekBack(): string {
    return $this->queue->getNthNode($this->queue->getSize())->data;
}

public function isEmpty(): bool {
    return $this->queue->getSize() == 0;
}
}

```

Skorzystamy teraz z naszej nowej klasy, aby sprawdzić operacje na kolejce dwustronnej:

```

try {
    $agents = new DeQueue(10);
    $agents->enqueueAtFront("Franek");
    $agents->enqueueAtFront("Janek");
    $agents->enqueueAtBack("Krzysiek");
    $agents->enqueueAtBack("Adrian");
    $agents->enqueueAtFront("Michał");
    echo $agents->dequeueFromBack() . "\n";
    echo $agents->dequeueFromFront() . "\n";
    echo $agents->peekFront() . "\n";
} catch (Exception $e) {
    echo $e->getMessage();
}

```

Jeśli przyjrzymy się temu przykładowi kodu, zauważymy, że najpierw jest w nim dodawany na początku kolejki element Franek, a potem również na początku element Janek. Sekwencja wygląda zatem teraz następująco: Janek, Franek. Następnie z tyłu kolejki dodajemy element Krzysiek, a potem Adrian. W wyniku tego otrzymujemy sekwencję: Janek, Franek, Krzysiek, Adrian. Na samym końcu kodu dodajemy element Michał na przodzie kolejki. Sekwencja przyjmuje zatem ostatecznie postać: Michał, Janek, Franek, Krzysiek, Adrian.

Jako że usuwanie z kolejki zaczynamy od tyłu, pierwszym elementem, który ją opuszcza, jest Adrian; następnie usuwamy element z przodu — jest nim Michał. Operacja podejrzenia elementu znajdującego się na początku kolejki spowoduje zwrócenie elementu Janek. Oto dane wyjściowe wygenerowane przez nasz kod po uruchomieniu:

```

Adrian
Michał
Janek

```


Podsumowanie

Stosy i kolejki są jednymi z najczęściej używanych struktur danych. Z tych abstrakcyjnych typów danych możemy w przyszłości korzystać na wiele różnych sposobów. W tym rozdziale poznaliśmy rozmaite metody implementowania stosów i kolejek, jak również różne rodzaje kolejek. W następnym zamierzamy omówić temat rekurencji będącej szczególnym sposobem rozwiązywania większych problemów poprzez ich podział na mniejsze jednostki.

Skorowidz

A

Abstract Data Types, *Patrz:* ADT
ADT, 22, 23, 127
algorytm, 19, 30
 analiza, 33, 34, 35, 36
 bazujący na szyfrze DES, 288
Bellmana-Forda, 224, 225, 226, 227
dane
 wejściowe, 30, 35
 wyjściowe, 30
Dijkstry, 221, 222, 223, 224, 227
dopasowania do wzorca, 252
dziel i zwyciężaj, 122, 171
Floyda-Warshalla, 218, 220, 221
iteracyjny, 110
Kahna, 216, 217, 218
KMP, 253, 254, 255
Knutha-Morrisa-Pratta, *Patrz:* algorytm KMP
kolorowania grafu, 197
Kruskala, 231
MD5, 287
mieszający, 288
Needlemana-Wunscha, 264
 implementacja, 264, 265, 266, 267
Prima, 228, 229, 230, 231
przypadek
 najgorszy, 35
 najlepszy, 35
 przeciętny, 35

quicksort, *Patrz:* algorytm sortowania szybkiego rekurencyjny, 109, 110, 122, 267
 złożoność, 122, 123
SHA1, 287
SHA256, 287
sortowania, 157
 bąbelkowego, 158, 159, 161, 165
 przez scalanie, 172
 przez wstawianie, 168
 przez wybieranie, 166
 szybkiego, 175
 złożoność, 190
wydajność, 33
wyszukiwania binarnego, 133, 147, 185, 187, 188
 powtarzalnego, 187
z nawrotami, 249, 267
zachłanny, 228, 231, 249, 255, 256, 260
 zastosowania, 258, 259
 złożoność, 34, 35
aplikacja częściowa, 301
atak typ brute force, 288

B

B-drzewo, 135, 155
Bellmana-Forda algorytm, *Patrz:* algorytm Bellmana-Forda
BFS, 194, 195, 202, 212
 grafu, 197, 212, 213
 implementacja, 195, 196
 złożoność, 197

biblioteka

 PHP DS, 288, 289
 standardowa PHP, *Patrz:* SPL
 Tarsana, 301, 302, 303, 304, 305
b-kopiec, 236
Bloom'a filtr, *Patrz:* filtr Bloom'a
BST, *Patrz:* drzewo binarne poszukiwań
bufor, 101

C

CF, 271
ciąg Fibonacciego, 111, 112, 250, 251, 260
collaborative filtering, *Patrz:* CF

D

DAG, 210, 216
dane
 abstrakcyjne, *Patrz:* ADT
 porządkowanie, *Patrz:* sortowanie
 sekwencyjne, 41
 skalarne, 25
 struktura, *Patrz:* struktura danych
 typ, *Patrz:* typ
DFS, 194, 198, 202, 212, 216
 implementacja, 195, 198, 199, 202, 214
 złożoność, 203

Dijkstry algorytm,
Patrz: algorytm Dijkstry
 domknięcie, 300
 dopasowanie do wzorca, 252
 drzewiec, 236
 drzewo, 24, 25, 28, 29, 40, 122, 127, 206
 AVL, 134, 155
 binarne, 132, 135
 doskonałe, 136
 implementacja, 136, 138
 kompletne, 136
 pełne, 135, 236, 237
 poszukiwań, 133, 135, 140, 141, 142, 143, 155
 poszukiwań
 samorównoważące się, 133, 134
 czerwono-czarne, 134, 155
 dopuszczalne,
 Patrz: drzewo AVL
 genealogiczne, 127
 głębokość, 129
 implementacja, 130, 305
 kategorii, 114
 tablica kategorii, 115
 zagnieżdżone
 wielopoziomowo, 114
 krawędź, *Patrz:* krawędź
 N-arne, 135
 odpowiedź na komentarz, 116
 poszukiwań, 194, 195
 przechodzenie, 129, 151
 bezpośrednie, 151
 implementacja, 153
 z opóźnieniem, 153
 z wyprzedzeniem, 152
 przeglądanie, *Patrz:* drzewo
 przechodzenie
 rozpinające minimalne,
 Patrz: MST
 wysokość, 129
 złożoność, 154
 dziecko, 128

F

Fibonacciego ciąg, *Patrz:* ciąg
 Fibonacciego
 filtr Blooma, 275, 276

filtrowanie wspólne, *Patrz:* CF
 Floyda-Warshalla algorytm,
 Patrz: algorytm Floyda-Warshalla
 FP, *Patrz:* programowanie
 funkcyjne
 fraktal, 107
 funkcja
 anonimowa, 300
 append, 304
 array_diff, 282
 array_intersect, 282
 array_map, 281
 array_merge_recursive, 126
 array_pop, 86, 280
 array_push, 94, 280
 array_rand, 282
 array_replace_recursive,
 126
 array_search, 280
 array_shift, 94, 282
 array_sum, 281
 array_unshift, 282
 array_walk_recursive, 125
 arsort, 179
 asort, 179
 base_convert, 284
 base64, 285
 base64_decode, 285
 BFS, 214
 bin2hex, 284
 bindec, 284
 crypt, 288
 current, 280
 częściowa aplikacja, 301
 czysta, 299
 decbin, 284
 dechex, 284
 decoct, 284
 end, 280
 hash, 194, 288
 hash_algos, 288
 hex2bin, 284
 hexdec, 284
 krsort, 179
 ksort, 179
 lambda, 300
 levenshtein, 285
 md5, 287
 metaphone, 287

mieszająca, 53, 193, 194, 276, 287
 natcasesort, 179
 natsort, 179
 next, 280
 octdec, 284
 password_hash, 288
 password_verify, 288
 pierwszej klasy, 299, 300
 prev, 280
 range, 182
 rekurencyjna, 110, 115, 125, 126
 repetitiveBinarySearch, 189
 reset, 280
 rozwijanie, 300, 301
 rozwinięta, 302
 rsort, 179
 search, 182
 sha1, 287
 shuffle, 282
 similar_text, 286
 skrótu, *Patrz:* funkcja
 mieszająca
 sort, 179
 sortująca, 179, 259
 soundex, 286
 str_split, 257
 strops, 252
 tablicowa, 52, 126, 280
 uasort, 179
 uksort, 179
 usort, 179, 259
 wbudowana, 284
 wywołania zwrotnego,
 125, 300
 wyższego rzędu, 299

G

GCD, *Patrz:* NWD
 gra w szachy, 267
 graf, 24, 25, 29, 45, 205, 206
 BFS, *Patrz:* BFS grafu
 kolorowanie, 197
 lista sąsiedztwa, 211
 macierz sąsiedztwa, 211, 213, 225
 nieskierowany, 29, 46, 209, 219

porządek topologiczny, 216
skierowany, 29, 209
acykliczny, *Patrz:* DAG
ścieżka, *Patrz:* ścieżka
ważony, 210, 218, 219
węzeł, *Patrz:* węzeł
Greatest Common Division,
Patrz: NWD

H

hashmapa, 39
haszowanie, *Patrz:* mieszanie
Hoare'a podział,
Patrz: podział Hoare'a
Huffmana kodowanie,
Patrz: kodowanie Huffmana
Hypertext Preprocessor,
Patrz: PHP

I

incydencja, 208
interfejs
Collection, 290
Hashable, 290
Iterator, 72
Sequence, 290
stosu, 84
iterator, 124, 125

J

język
Clojure, 297
Elixir, 297
Erlang, 297
funkcyjny, 297
Haskell, 297
imperatywno-obiektowy,
297
PHP, *Patrz:* PHP
Scala, 297
skryptowy, 20
słabo typowany, 23

K

Kahna algorytm,
Patrz: algorytm Kahna
klasa
DS\Deque, 293
DS\Map, 291
DS\Queue, 293, 294
DS\Set, 291
DS\Stack, 293
DS\Vector, 290, 293
MaxHeap, 242
SPL, 283, 288
SplDoublyLinkedList, 37,
81, 90, 283
metody, 81
SplFixedArray, 37, 47, 48,
49, 50, 53, 57, 283, *Patrz*
też: tablica SplFixedArray
tworzenie, 50
wydajność, 51
SplHeap, 37, 248, 283
SplMaxHeap, 37, 248, 283
SplMinHeap, 37, 248, 283
SplObjectStorage, 37, 283
SplPriorityQueue, 37, 99,
283
SplQueue, 37, 96, 283
SplStack, 37, 90, 283
sterty, 238
klucz, 42, 53, 193, 257, 283
skrót, 288
Soundex, 286
zaburzający, 288
Knutha-Morrisa-Pratta
algorytm, *Patrz:* algorytm
KMP
kod, 32
kodowanie Huffmana, 255, 256
kolejka, 21, 23, 24, 25, 27, 37,
40, 93, 195, 196, 197, 200,
202, 215, 279, 283, 293
cykliczna, 100, 101
dwustronna, 294
implementacja, 102
implementacja, 94, 95, 304
podwójna, 102

priorytetowa, 24, 25, 37, 96,
221, 222, 223, 224, 235,
242, 256, 294
implementacja, 97, 99,
100, 242, 243, 244
wydajność, 294
kolekcja, 40, 279
par klucz-wartość, 291
kompresja, 256
kopiec, *Patrz:* sterta
korelacja Pearsona, 271, 272, 274
korzeń, 28, 128, 131, 145
krawędź, 29, 128, 205, 206
incydentna, 208
nieskierowana, 207, 209
skierowana, 206, 209, 216
ważona, 207, 210, 219
Kruskala algorytm, *Patrz:*
algorytm Kruskala
kryptografia, 287, 288

L

las, 129, 231
LCS, 262, 263, 264
lista, 24, 40, 59
cykliczna, 63
tworzenie, 73
dwukierunkowa, 25, 26, 37,
63, 283
operacje, 75
tworzenie, 75
złożoność czasowa, 80
element
pobieranie, 70
usuwanie, 60, 65, 67, 68,
69, 76, 78, 79
wstawianie, 60, 65, 66,
67, 75, 76, 77, 78
wyszukiwanie, 60, 64,
65, 69, 76
iterowalna, 73
jednokierunkowa, 25, 26,
59
nieuporządkowana, 182
odwracanie, 60, 65, 69
operacje, 60
posortowana, 183, 190
malejąco, 187

lista
 uporządkowana, *Patrz:* lista
 posortowana
 wielokierunkowa, 63, 64
 wielokrotnie połączona, 64
 złożoność czasowa, 71

liść, 128, 144, 149

longest common subsequence,
Patrz: LCS

Longest Proper Prefix Suffix,
Patrz: LPS

LPS, 254

L

łańcuch

funkcji, 300, 302

znakowy, 42, 283

klucz skrótu, 288

klucz Soundex, 286

kompresja, 256

podobieństwo, 285, 286,
 287

przeszukiwanie, 252, 253

skrót, 288

skrót MD5, 287

skrót SHA1, 287

standard MIME, 285

weryfikacja, 288

łącze, 26, 59

M

macierz rzadka, 274, 275

mapa, 24, 25, 28, 37, 279, 291

mieszająca, 48

uporządkowana, 40

mechanizm wspólnej filtracji,
Patrz: CF

memoizacja, 249, 250, 252, 260

menu, 114

metoda

create, 239

current, 72

dziel i zwyciężaj, *Patrz:*
 algorytm dziel i zwyciężaj

extractMax, 244

extractMin, 239

fromArray, 51

key, 72

next, 72

rewind, 72

setSize, 52

siftDown, 244

siftUp, 239, 244

valid, 72

mieszanie, 287, 288

model

FIFO, 24, 27, 93, 242

LIFO, 24, 27, 83, 200, 202

MST, 227, 228, 231

N

najdłuższy właściwy
 prefikso-sufiks, *Patrz:* LPS

najdłuższy wspólny podciąg,
Patrz: LCS

Największy Wspólny Dzielnik,
Patrz: NWD

Needlemana-Wunscha
 algorytm, *Patrz:* algorytm
 Needlemana-Wunscha

notacja dużego O, 35, 36, 123,
 127

NWD, 111

O

obiekt, 23

overflow exception,
Patrz: wyjątek przepełnienia

P

Pearsona korelacja,
Patrz: korelacja Pearsona

PECL, 279, 289

pętla

for, 41, 44, 110, 182

foreach, 44, 73

while, 44, 62, 72, 110, 140

PHP, 279, 297

biblioteka

PHP DS, 288, 289

standardowa, *Patrz:* SPL

repozytorium rozszerzeń,
Patrz: PECL

rozszerzenie DS, 288

poddrzewo, 129

podział Hoare'a, 175

potomek, 128

Prima algorytm,
Patrz: algorytm Prima

problem, 30

bioinformatyczny, 264

definicja, 30

plecakowy dyskretny, 261, 262

programowanie

dynamiczne, 249, 260

podejście oddolne, 260

podejście odgórne, 260

zastosowania, 260, 261,
 262, 264

funkcyjne, 297, 298, 299, 301

cechy, 298

zorientowane

na wartości, 298

obiektoowo, 37

przeszukiwanie, *Patrz też:*
 wyszukiwanie

w głąb, *Patrz:* DFS

wszerz, *Patrz:* BFS

przodek, 128

pseudokod, 31

przekształcanie w
 prawdziwy kod, 32

Q

quicksort, *Patrz:* algorytm
 sortowania szybkiego,
 sortowanie szybkie

R

rekomendacja, 271, 274

rekurencja, 107, 108, 110, 115,
 120, 130, 131, 140, 185, 260,
 267

binarna, 112

cykl, 109

głębokość, 123

liniowa, 112, 122

ogonowa, 112

przypadek bazowy, 109, 124

wzajemna, 113

zagnieżdżona, 113

rodzeństwo, 128
rodzic, 128, 148

S

sekwencja

nieuporządkowana, 97
uporządkowana, 97

sekwencjonowanie, 266

DNA, 264

silnia, 108

słownik, 28, 40, 279, *Patrz też:*

mapa

sortowanie, 157, *Patrz też:*

algorytm sortowania

bąbelkowe, 158

implementacja, 159, 162,
165

optymalizacja, 161

złożoność, 161

kubelkowe, 158

przez kopcowanie, 235, 245

złożoność, 247

przez scalanie, 158, 171

implementacja, 173

złożoność, 174

przez wstawianie, 158, 168

implementacja, 170

złożoność, 171

przez wybieranie, 158, 165

implementacja, 167

złożoność, 167

szybkie, 158, 175

implementacja, 176

oś, 175

partycjonowanie, 175

podział, 175

złożoność, 178

topologiczne, 216

sól, 288

SPL, 37, 81, 99, 124

Standard PHP Library, *Patrz:*

SPL

sterta, 25, 29, 37, 100, 235, 283

binarna, 236

implementacja, 237, 240,
241

złożoność, 241

element

usuwanie, 236

wstawianie, 236

zamiana miejscami, 236,
237

Fibonacciego, 221, 231, 236

korzeń, 235, 236

luźna, 236

potrójna, 236

tworzenie, 236, 245

typ

max, 29, 37, 235, 248

min, 29, 37, 221, 235, 240,
248, 257

wydobywanie

maksimum, 236, 237

minimum, 236, 237, 239

stopień, 128

stos, 23, 24, 25, 27, 37, 40, 83,
200, 202, 279, 283, 293

implementacja, 84, 86, 88,
303

interfejs, *Patrz:* interfejs

stosu

wierzchołek, 27, 83

zastosowania, 90, 91

złożoność

czasowa, 87

pamięciowa, 88

stóg, *Patrz:* sterta

struktura

danych, 19, 20, 21, 45

hierarchiczna, 127

implementacja, 54

liniowa, 24, 127

nieliniowa, 24, 28, 29

odpowiedź na komentarz,
116

sudoku, 267

Ś

ścieżka, 128, 208, 215

najkrótsza, 218, 219, 227

T

tablica, 23, 24, 25, 37, 40, 237

asocjacyjna, 24, 40, 42, 53,
193, 257, *Patrz też:* mapa

bitowa, 276

element, 280, 282, 282

jednowymiarowa, 26

liczbowa, 40, 41, 42

mapowanie, 281

mieszająca, 40, 53, 127,
193, 279

nieposortowana, 190

o stałym rozmiarze, 47, 52

odejmowanie, 282

PHP, 39, 40, 57, 127, 193,
279, 280

implementacja struktury,
54

implementacja zbioru, 55

struktura wewnętrzna, 49

wydajność pamięciowa,
49, 57

przekazywanie przez
referencję, 170, 280

skojarzeniowa, *Patrz:* mapa
SplFixedArray, 57

przekształcenie w tablicę
PHP, 52

tworzenie, 47, 51

zmiana rozmiaru, 52

tworzenie, 40

wielowymiarowa, 40, 43,
44, 45

tworzenie, 53

wskaźnik bieżącego miejsca,
280

Theunissen Rudi, 288

typ

abstrakcyjny, *Patrz:* ADT

całkowity, 23

deklarowanie, 23

logiczny, 23

null, *Patrz:* null

statyczny, 23

tekstowy, 23

zmiennoprzecinkowy, 23

U

uczenie maszynowe, 249

underflow exception, *Patrz:*
wyjątek niedoboru

W

Watkins Joe, 288

wektor, 40, 279, 290,

Patrz też: lista

węzeł, 26, 28, 45, 59, 205,

Patrz też: lista element

bez dzieci, *Patrz:* liść

klucz, 129

końcowy, 128

potomny, 128

szczytowy, 128

tworzenie, 61, 132

usuwanie, 142, 147, 149,

150

wewnętrzny, 128

wstawianie, 141

wysokość, 129

wyszukiwanie, 141

zawartość, 60

zewewnętrzny, 128

wierzchołek, 29, 205, 206

incydentny, *Patrz:*

incydencja

sąsiadujący, 207, 211

stopień

wchodzący, 208, 216

wychodzący, 208

wyjątek

E_NOTICE, 288

niedoboru, 85

przepelnienia, 85

RuntimeException, 47

wykonanie częściowe, 301

wyrażenie regularne, 252

wyszukiwanie, 181,

Patrz też: przeszukiwanie

binarne, 122, 183, 184

implementacja, 184, 186

powtarzalne, 187, 188, 189

złożoność, 187, 190

interpolacyjne, 191

liniowe, 181, 182

przy użyciu tablicy

mieszającej, 193

sekwencyjne, 30, 190,

Patrz też:

wyszukiwanie liniowe

w drzewach, 195

w głąb, *Patrz:* DFS

wszerz, *Patrz:* BFS

wykładnicze, 192

Z

zapytanie bazodanowe, 275

zasoby, 23, 33

zbiór, 24, 25, 28, 279, 291

implementacja, 55

uporządkowany, 28

zmienna, 25

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Testy penetracyjne — klucz do bezpieczeństwa Twojej aplikacji!

Algorytmy i struktury danych leżą u podstaw programowania. Zrozumienie zasad rządzących tymi zagadnieniami jest koniecznym warunkiem opracowania prawidłowej i efektywnej aplikacji. Niestety, wielu programistów uznaje tę tematykę za zbyt złożoną czy zbyt banalną i nie poświęca jej wystarczającej uwagi. Takie podejście często się mści: modne narzędzia, frameworki czy technologie deweloperskie nie zapewnią sukcesu, jeśli projektant nie przemyśli zastosowanych algorytmów i struktur danych. Z tego obowiązku nie zwalniają nawet narzędzia wbudowane w język PHP!

Jeśli chcesz biegle posługiwać się algorytmami, wzięteś do ręki właściwą książkę! Przedstawiono tu podstawy implementacji algorytmów i struktur danych w PHP, dzięki czemu poznasz rodzaje struktur i powody, dla których warto je wybierać, a także dowiesz się, gdzie i kiedy należy stosować poszczególne algorytmy. Znajdziesz tu dużo praktycznych przykładów, które uzupełniono rysunkami i wyczerpującym komentarzem. Przystępne i zrozumiałe wyjaśnienia ułatwią Ci szybkie przyswojenie prezentowanych koncepcji, nawet tak złożonych, jak programowanie dynamiczne, algorytmy zachłanne, algorytmy z nawrotami czy funkcyjne struktury danych.

Najważniejsze zagadnienia:

- podstawy analizy algorytmów i struktur danych
- tablice, listy i drzewa
- stosy, kolejki i algorytmy rekurencyjne
- sortowanie, wyszukiwanie, sterty i kopce
- wsparcie ze strony PHP, w tym biblioteki PECL i Tarsana

Mizanur Rahman od 14 lat rozwija aplikacje w PHP; znawca Laravela, CodeIgnitera, Symfony, JavaScriptu, C, C++, Javy, Node.js, Socket.IO i React.js. Jest właścicielem dwóch startupów technologicznych. Aktywnie angażuje się w życie kilku społeczności programistycznych, takich jak PHPXperts, Agile Bangladesh czy Project Euler. Regularnie wygłasza referaty na różnych konferencjach i seminariach technologicznych. Wraz z żoną Nishą i dwoma synami, Adiyanem i Mikhaelem, mieszka w Dhace w Bangladeszu. Jego pasją są podróże po świecie.

	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI <i>Sięgnij po więcej!</i>	
 helion.pl		ISBN 978-83-283-4085-5	
 0 801 339900	AKADEMIA IT & BUSINESS		
 0 601 339900	WWW.SZKOLENIA.HELION.PL	9 788328 340855	
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 59,00 zł	

Packt