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

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

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.

Spis treści

ROZDZIAŁ 4

Konstruowanie stosów i kolejek

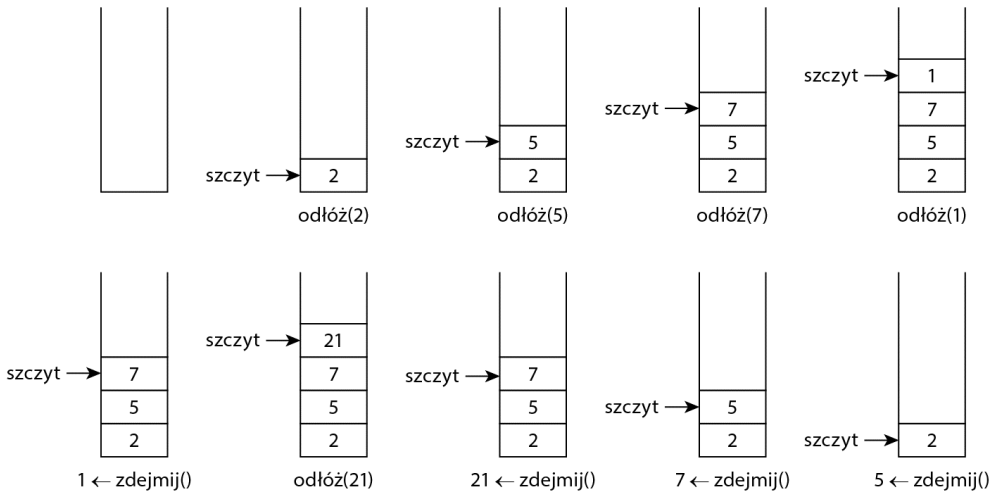
W codziennej praktyce najczęściej używamy dwóch najbardziej popularnych struktur danych. Możemy założyć, że są one wzorowane na obiektach istniejących w prawdziwym świecie, mają jednak ogromny wpływ przede wszystkim na świat komputerowy. Mówimy tu o stosie (ang. *stack*) i kolejce (ang. *queue*). W stosy 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 zwykłe 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);
    }
}
```

Przeanalizujmy 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. Napiszmy 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 obsłużyć 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
pop	O(1)
push	O(1)
top	O(1)
isEmpty	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 spróbowaliśmy 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:

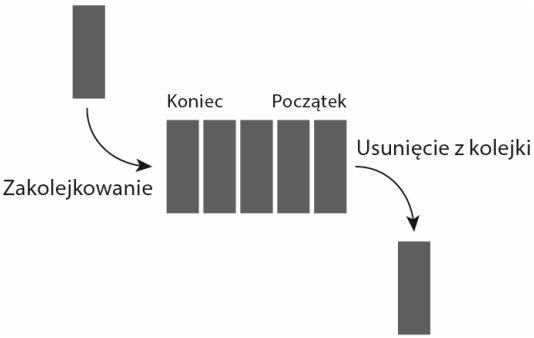
```
$expressions = [];
$expressions[] = "8 * (9 -2) + { (4 * 5) / ( 2 * 2) }";
$expressions[] = "5 * 8 * 9 / ( 3 * 2 ) )";
$expressions[] = "[{ (2 * 7) + ( 15 - 3) ]";

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 utwórzmy 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 pojawią 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 kolejkowania 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 węźle 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 uwzględniać 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 uwzględniać 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();
}
```

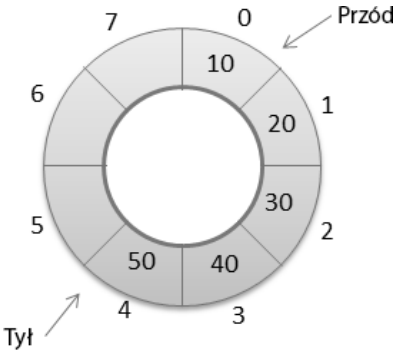
Przedstawiony powyżej kod wygeneruje te same dane wyjściowe co przykład wykorzystujący listę. Dodatkową przewagą rozwiązania polegającego na użyciu klasy `MyPQ` rozszerzającej klasę `SplPriorityQueue` jest to, że możemy tu wybrać sposób sortowania elementów (tj. to, czy mają być one ułożone w kolejności 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 stercie, 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 bufory cykliczne (ang. *circular buffers*) lub bufory 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:

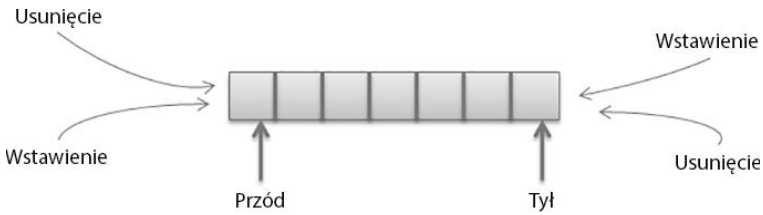
```
class CircularQueue implements Queue {

   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

F

G

L

M

N

O

P

Q

R

W

Z

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, wziąłeś 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.

ISBN 978-83-283-4085-5