

# Эффективные фундаментальные структуры данных в PHP7

PHP имеет всего одну структуру данных для управления всем. array – сложный, гибкий, гибридный, сочетает в себе поведение list и linked map. Но мы используем его для всего, потому что PHP придерживается **прагматичного подхода**: иметь предельно правильный, здравый и реалистичный способ решения проблемы, исходящий из практических, а не теоретических рассуждений. array позволяет делать работу, хотя о нем и так много рассказывают на лекциях по информатике. Но, к сожалению, с гибкостью приходит и сложность.

Последний релиз PHP вызвал большое оживление в сообществе. Мы не могли дождаться того, чтобы начать использовать новые возможности и почувствовать вкус ~2x прироста производительности. Одна из причин, почему это случилось – структура array была переработана. Но массивы все также придерживаются принципа «оптимизировано для всего; оптимизировано для ничего», еще не все идеально, есть возможности для совершенствования.

*А что насчет структур данных SPL?*

К сожалению... они ужасны. Раньше, до PHP7, они предлагали некоторые преимущества, но сейчас мы дошли до точки, когда использование SPL не имеет практического смысла.

*Почему мы не можем просто поправить и улучшить их?*

Да, мы могли бы, но я считаю, что их дизайн и реализация настолько бедны, что лучше бы найти более современную замену.

**«SPL data structures are horribly designed.»**

– Anthony Ferrara

---

**Введение:** php-ds – расширение для PHP7, добавляющее структуры данных. Этот пост кратко охватывает поведение, производительность и преимущества каждой из них. Также в конце вы найдете список ответов на ожидаемые вопросы.

**Github:** <https://github.com/php-ds>

**Пространство имен:** Ds\

**Интерфейсы:** Collection, Sequence, Hashable

**Классы:** Vector, Deque, Stack, Queue, PriorityQueue, Map, Set

---

## Collection (Коллекция)

Collection – это базовый интерфейс, охватывающий общую функциональность: `foreach`, `echo`, `count`, `print_r`, `var_dump`, `serialize`, `json_encode`, и `clone`.

## Sequence (Последовательность)

Sequence описывает поведение элементов, организованных в единый, линейный размер. В некоторых языках такая структура называется List (список). Подобен array, который использует инкрементальные ключи, за исключением некоторых особенностей:

- Значения **всегда** должны быть индексированы как [0, 1, 2, ..., size - 1]
- Извлечение или добавление приводит к обновлению индекса всех последовательных значений
- Поддерживает доступ к значениям только из индекса [0, size - 1]

## Варианты использования

- Везде, где вы бы хотели использовать array как список

(без ключей)

- Более эффективная альтернатива `SplDoublyLinkedList` и `SplFixedArray`

## Vector (Вектор)

Vector представляет собой Sequence, объединяющую значения в непрерывный буфер, увеличивающийся и уменьшающийся автоматически. Это наиболее эффективная последовательная структура данных, поскольку индекс элемента является прямым отражением его индекса в буфере, и увеличение вектора никак не повлияет на производительность.

### Сильные стороны

- Очень маленькое потребление памяти
- Очень быстрые итерации
- `get`, `set`, `push` и `pop` имеют сложность  $O(1)$

### Недостатки

- `insert`, `remove`, `shift` and `unshift` имеют сложность  $O(n)$

*Структурой номер один в Photoshop были Вектора.*

– **Sean Parent**, *CppCon 2015*

## Deque (Двусвязная очередь)

Deque (произносится как «deck») – это последовательность значений, объединенных в непрерывный буфер, увеличивающийся и уменьшающийся автоматически. Название является общепринятым сокращением от «*double-ended queue*». Используется внутри `Ds\Queue`.

Два указателя используется для отслеживания головы и хвоста. Наличие указателей позволяет изменять конец и начало буфера без необходимости перемещать другие элементы для освобождения места. Это делает `shift` и `unshift` настолько быстрым, что даже `Vector` не может конкурировать с этим.

Доступ к значению по индексу требует вычисления соответствующей позиции в буфере:  $((\text{head} + \text{position}) \% \text{capacity})$ .

## Сильные стороны

- Очень маленькое потребление памяти
- `get`, `set`, `push`, `pop`, `shift` и `unshift` имеют сложность  $O(1)$

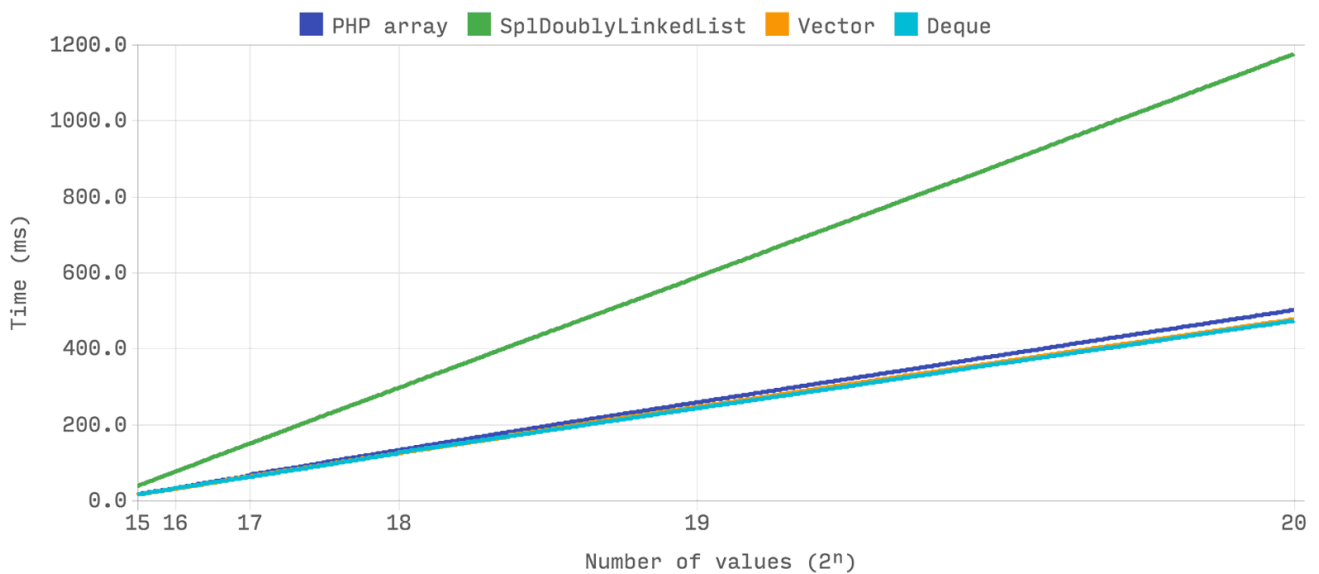
## Недостатки

- `insert`, `remove` имеют сложность  $O(n)$
- Емкость буфера должна иметь степень двойки ( $2^n$ )

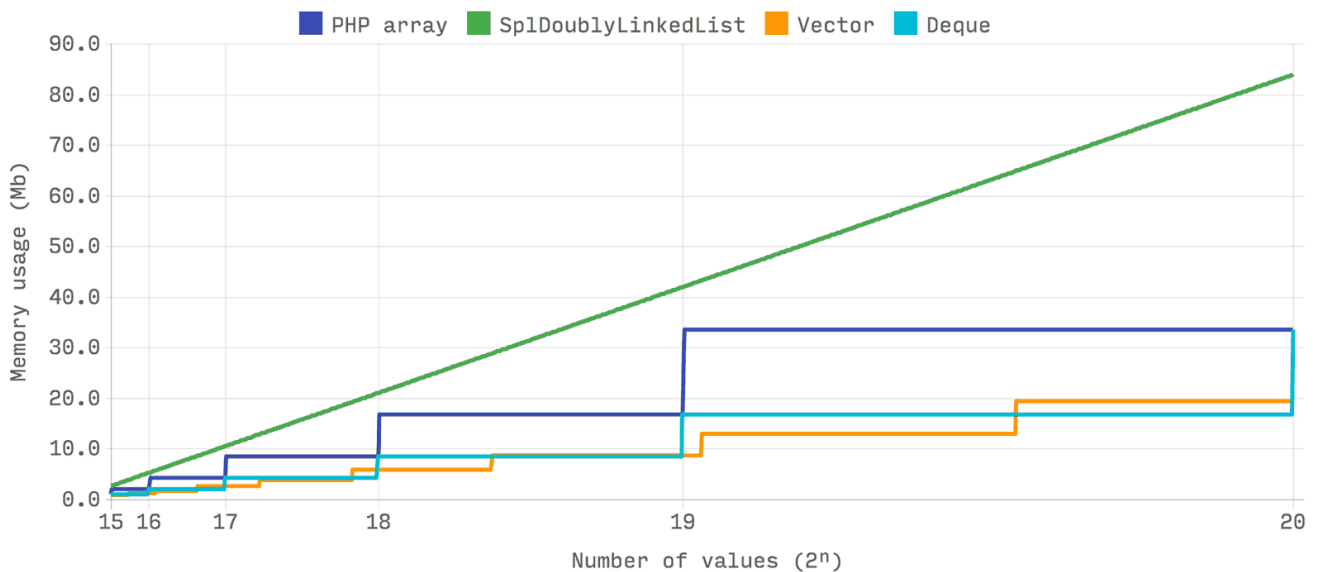
Следующий бенчмарк показывает общее затраченное время и память, используемую для операции `push`  $2^n$  случайных чисел. `array`, `Ds\Vector` и `Ds\Deque` обрабатывают быстро, но `Sp1DoublyLinkedList` стабильно показывает результат **более чем в 2 раза хуже**.

`Sp1DoublyLinkedList` выделяет память для каждого значения по отдельности, поэтому и происходит ожидаемый рост по памяти. `array` и `Ds\Deque` при своей реализации выделяют память порционно для поддержания достаточного объема для  $2^n$  элементов. `Ds\Vector` имеет фактор роста 1.5, что влечет за собой увеличение количества выделений памяти, но меньший расход в целом.

Sequence::push (Time taken)

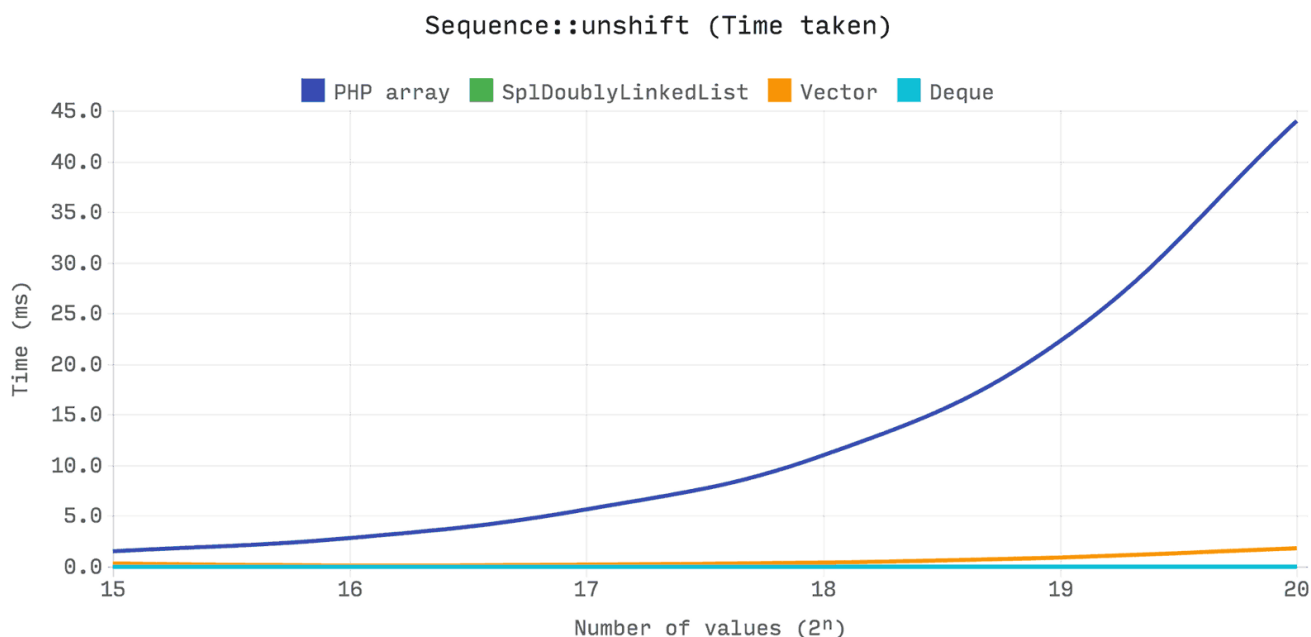


Sequence::push (Memory usage)



Следующий бенчмарк показывает время, затраченное на `unshift` **единственного элемента** в последовательности значений размером  $2^n$ . Время, требующееся на установку значений не учитывается.

На графике видно, что `array_unshift` имеет сложность  $O(n)$ : всякий раз, когда объем выборки удваивается, растет и время, необходимое для `unshift`. Это объясняется тем, что каждый числовой показатель в диапазоне `[1, size - 1]` должен быть обновлен.



Но и `Ds\Vector::unshift` также  $O(n)$ , так почему же он намного быстрее? Имейте в виду, что `array` хранит каждое значение в `bucket` вместе с его ключем и хэшем. Поэтому приходится проверять каждый элемент и обновлять хэш, если индекс является числовым. На самом деле `array_unshift` выделяет новый массив для этого и заменяет старый, когда все значения скопированы.

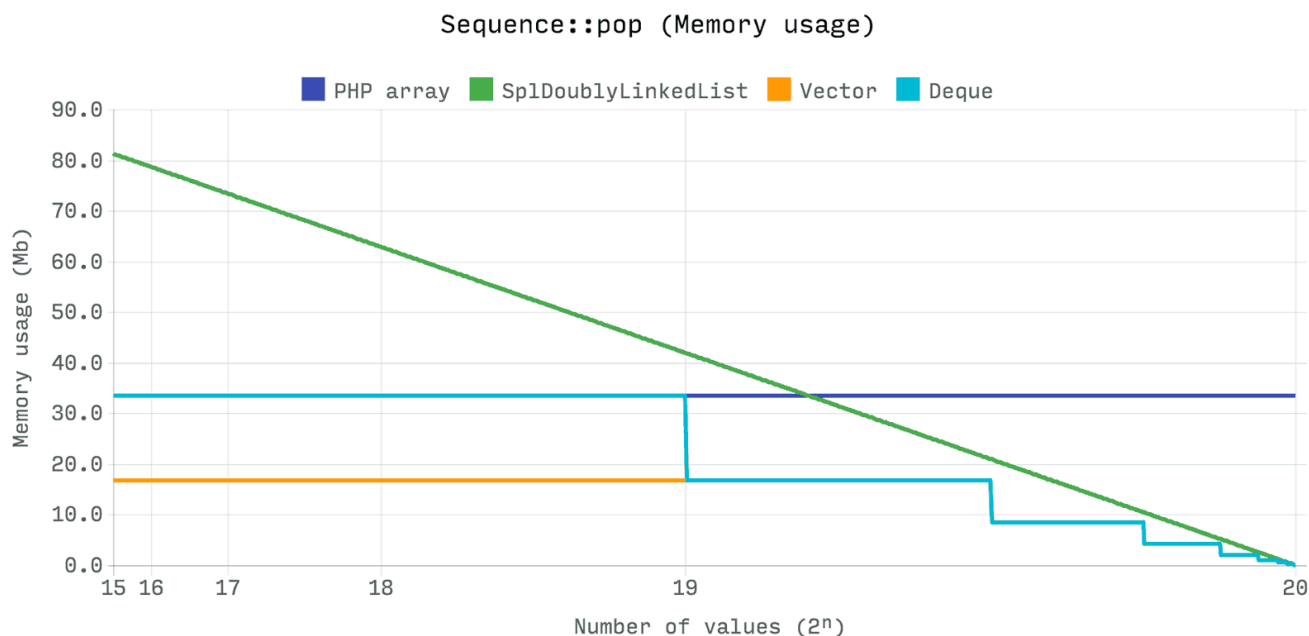
В векторе же индекс значения – это прямое отображение его индекса в буфере, поэтому все, что нам нужно сделать – сдвинуть каждое значение в диапазоне `[1, size - 1]` вправо на одну позицию. Делается это при помощи всего одной операции `memmove`.

`Ds\Deque` и `SplDoublyLinkedList` в свою очередь очень быстры, потому что на время для `unshift` значения не влияет размер выборки, т.е. его сложность будет  $O(1)$ .

На следующем тесте видно сколько памяти используется при  $2^n$  `pop` операций. Другими словами при изменении размера от  $2^n$  до нуля

Интересно тут то, что `array` всегда держит выделенную память, даже если его размер существенно уменьшается. `Ds\Vector` and `Ds\Deque` позволяют в два раза уменьшить выделяемые ресурсы,

если их размер падает ниже четверти своего текущего потенциала. `SplDoublyLinkedList` освобождает память после каждого удаления из выборки, поэтому мы можем наблюдать линейное снижение.

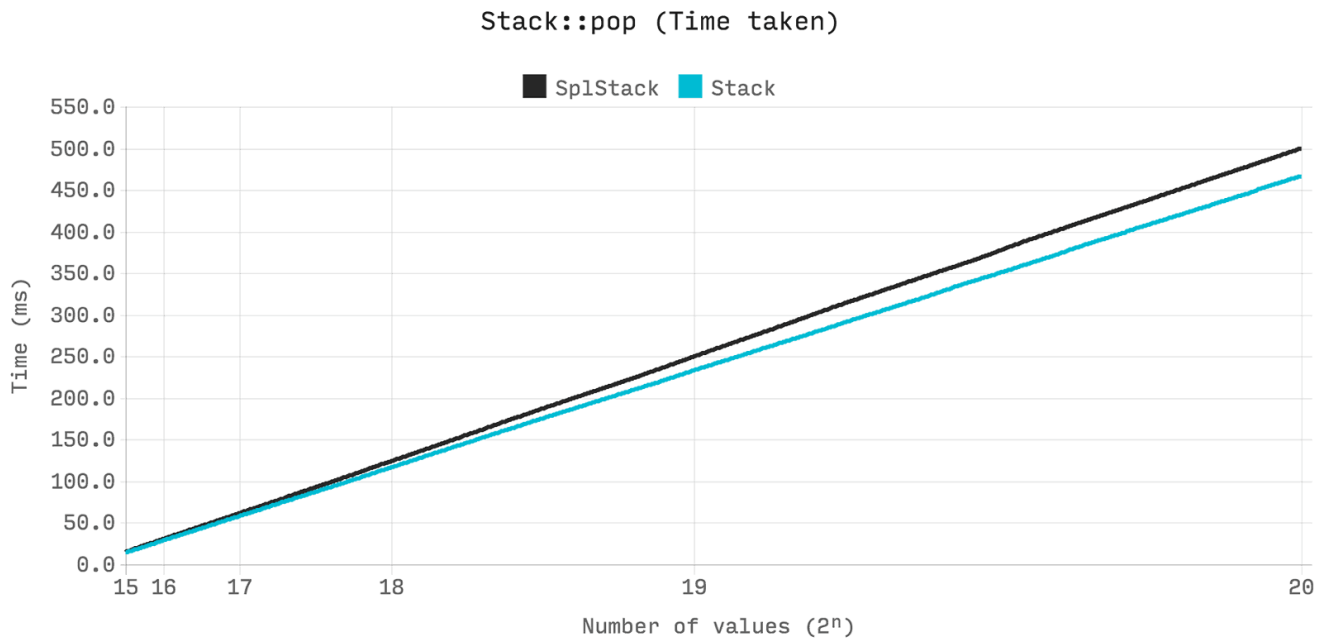


## Stack (Стек)

Стек – является коллекцией, организованной по принципу «последним пришёл – первым вышел» или «LIFO» (*last in – first out*), позволяющей получить доступ только к значению на вершине структуры. Вы можете думать о нем как об оружейном магазине с динамической емкостью.

`Ds\Stack` использует внутри себя `Ds\Vector`.

`SplStack` наследуется от `SplDoublyLinkedList`, поэтому производительность будет эквивалентна сравнению `Ds\Vector` to `SplDoublyLinkedList` из предыдущих тестов. Посмотрим на время, необходимое для выполнения  $2^n$  pop-операций, изменения размера от  $2^n$  до нуля.



## Queue (Очередь)

Очередь – тип данных с парадигмой доступа к элементам «*первый пришел – первый вышел*» («*FIFO*», «*First In – First Out*»). Такая коллекция позволяет получить доступ к элементам в порядке их добавления. Ее название говорит само за себя, представьте себе структуру как линию людей, стоящих в очереди на кассу в магазине.

Ds\Queue использует внутри себя DsDeque. SplQueue наследуется от SplDoublyLinkedList, поэтому производительность будет эквивалентна сравнению DsDeque с SplDoublyLinkedList, показанному в предыдущем бенчмарке.

## PriorityQueue (Очередь с приоритетом)

Очередь с приоритетом очень похожа на простую очередь. Элементы помещаются в очередь с указанным приоритетом и значение с наивысшим приоритетом всегда будет в передней части. Прямой перебор очереди с приоритетом очень



деструктивен, это будет последовательный вызов операций pop, что является очень затратной операцией.

Реализация очереди с приоритетом использует *max-heap*.

Принцип «*первый пришел – первый вышел*» сохраняется для значений с одинаковым приоритетом, так что группа значений с равным приоритетом можно рассматривать как обычную очередь.

А что же с производительностью? Следующий бенчмарк показывает время и память, требующиеся для операции push 2<sup>n</sup> случайных чисел со случайным приоритетом в очередь. Те же случайные числа будут использоваться для каждого из тестов. В тесте для Queue также генерируется случайный приоритет, хотя он и не используется.

Это, наверное, самый значимый из всех бенчмарков. Ds\PriorityQueue работает **более чем в два раза быстрее** чем SplPriorityQueue и использует только **5%** от его памяти – это в **20 раз более эффективное решение по памяти**.

Но как? Как может получиться настолько большая разница, когда SplPriorityQueue использует аналогичную внутреннюю структуру? Все сводится к тому, как хранятся значения в паре с приоритетом. SplPriorityQueue позволяет использовать любой тип значения для использования в качестве переменной, это приводит к тому, что в каждой паре приоритет занимает **32 байта**.

Ds\PriorityQueue поддерживает только целочисленные приоритеты, поэтому каждой паре выделяется **24 байта**. Но это все еще недостаточная разница для объяснения результата.

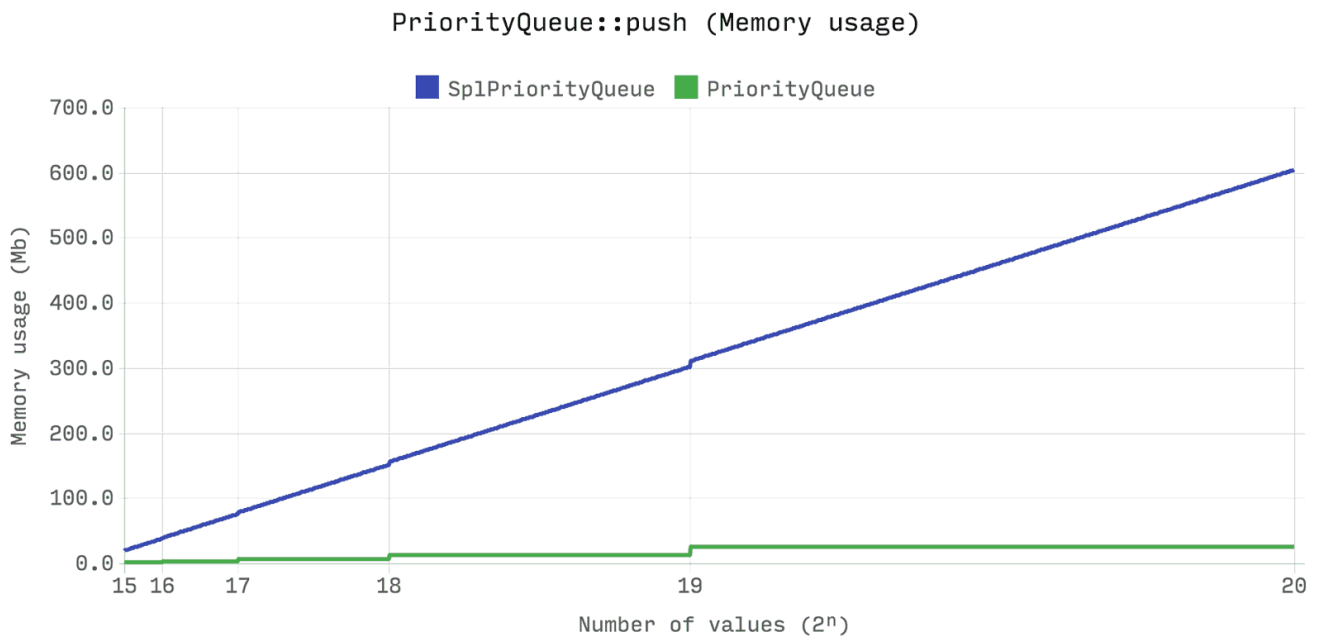
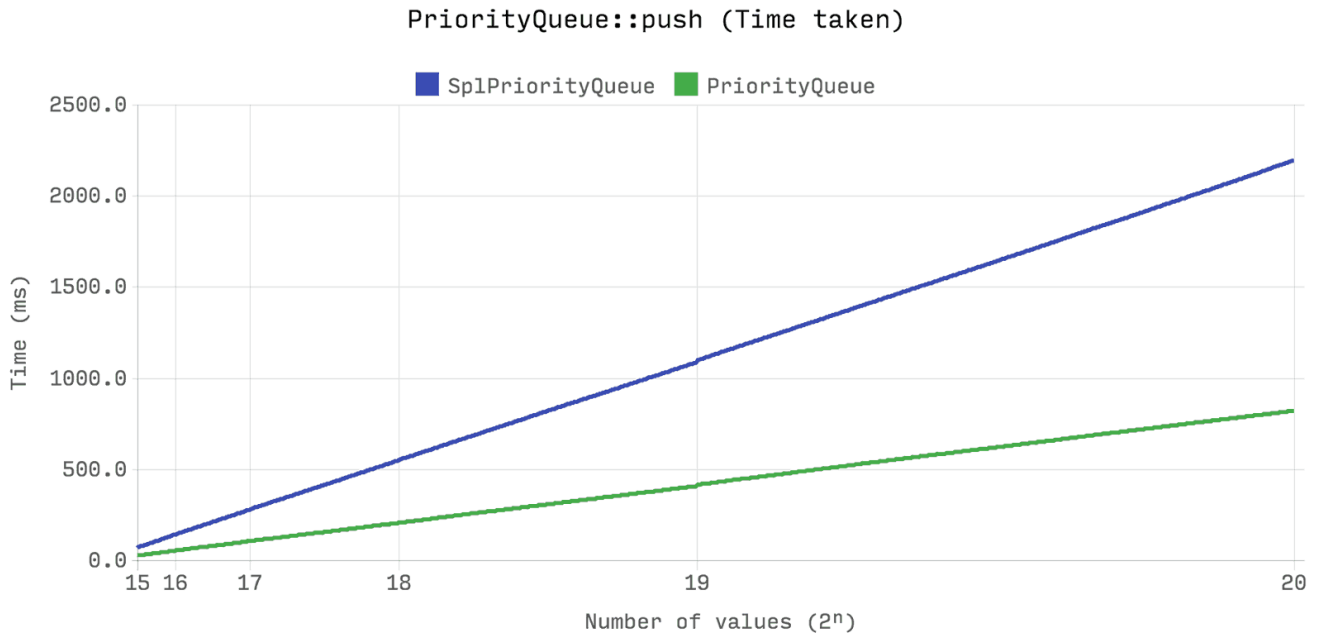
Если вы посмотрите на исходный код SplPriorityQueue::insert, то заметите, что он **инициализирует массив** для **хранения пары значение-приоритет**.

Т.к. массив имеет минимальную емкость 8, то для каждой пары на самом деле выделяется  $zval + HashTable + 8 * (Bucket + hash) + 2 * zend\_string + (8 + 16) \text{ byte string payloads} = 16 + 56 + 36$

\*  $8 + 2 * 24 + 8 + 16 = 432$  **байта** (64 бит).

### «Так... почему же массив?»

SplPriorityQueue использует ту же внутреннюю структуру SplMaxHeap, которая требует от значения быть типом zval. Очевидный (но неэффективный) способ создания zval-пары, т.к. zval сам используется как array.



# Hashable

Интерфейс, позволяющий объектам **быть использованными в качестве ключей**. Это альтернатива `spl_object_hash`, который детерминирует объект в хэш, базирующийся на его `handle`. Это означает, что два объекта, которые считались бы равными при сравнении, не имели бы равный хэш, т.к. они не являются одним и тем же экземпляром.

Hashable вводит только два метода: `hash` и `equals`. Многие другие языки поддерживают это изначально: в Java – `hashCode` и `equals`, или в Python `__hash__` и `__eq__`. Было несколько RFC, добавляющих подобное поведение и в PHP, но ни один из не был принят.

Все структуры, будут возвращать `spl_object_hash`, если ключи объектов, хранящиеся в них не реализуют в себе Hashable.

Структуры данных, работающие с интерфейсом Hashable: Map и Set.

## Map (Ассоциативный массив)

Map является последовательной коллекцией пар ключ-значение, практически идентичной `array` в аналогичном контексте. **Ключи могут быть любого типа**, единственное условие – уникальность. При повторном добавлении ключа значения заменяются.

Как и в `array`, порядок вставки сохраняется.

### Сильные стороны

- Производительность и эффективность использования памяти **практически идентичны** `array`
- Автоматическое освобождение памяти при уменьшении размера
- Ключи и значения могут быть любого типа, включая объекты
- Поддерживает работу с объектами, реализующими интерфейс

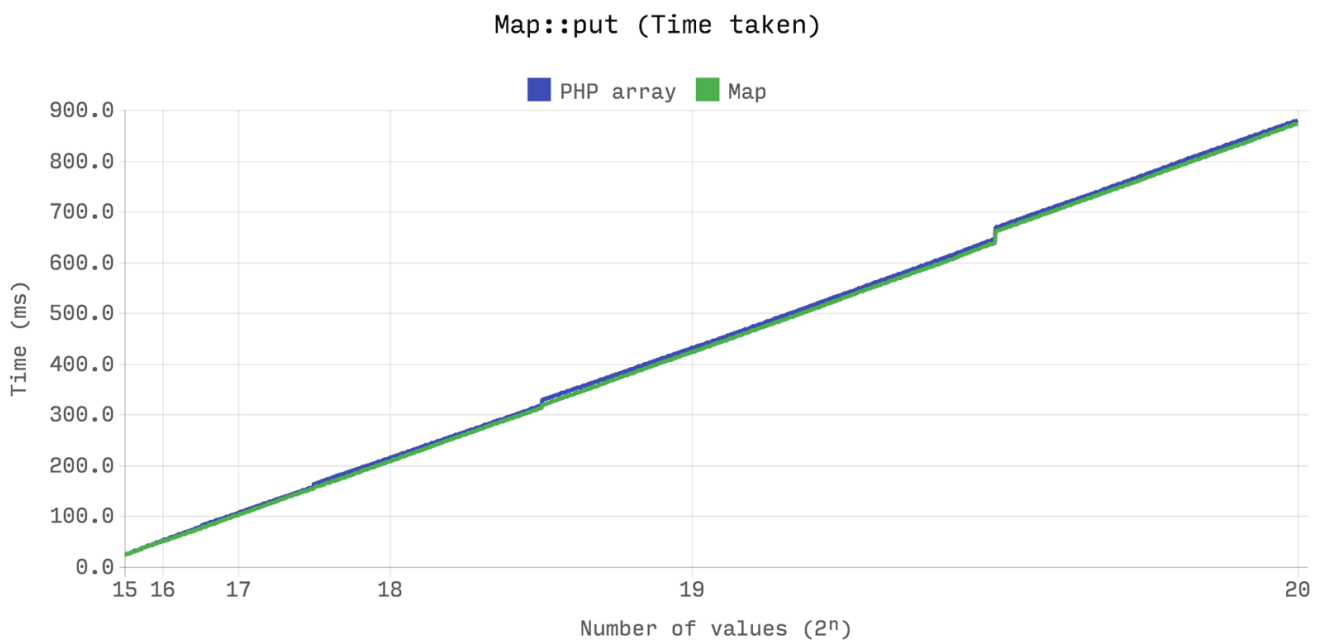
Hashable

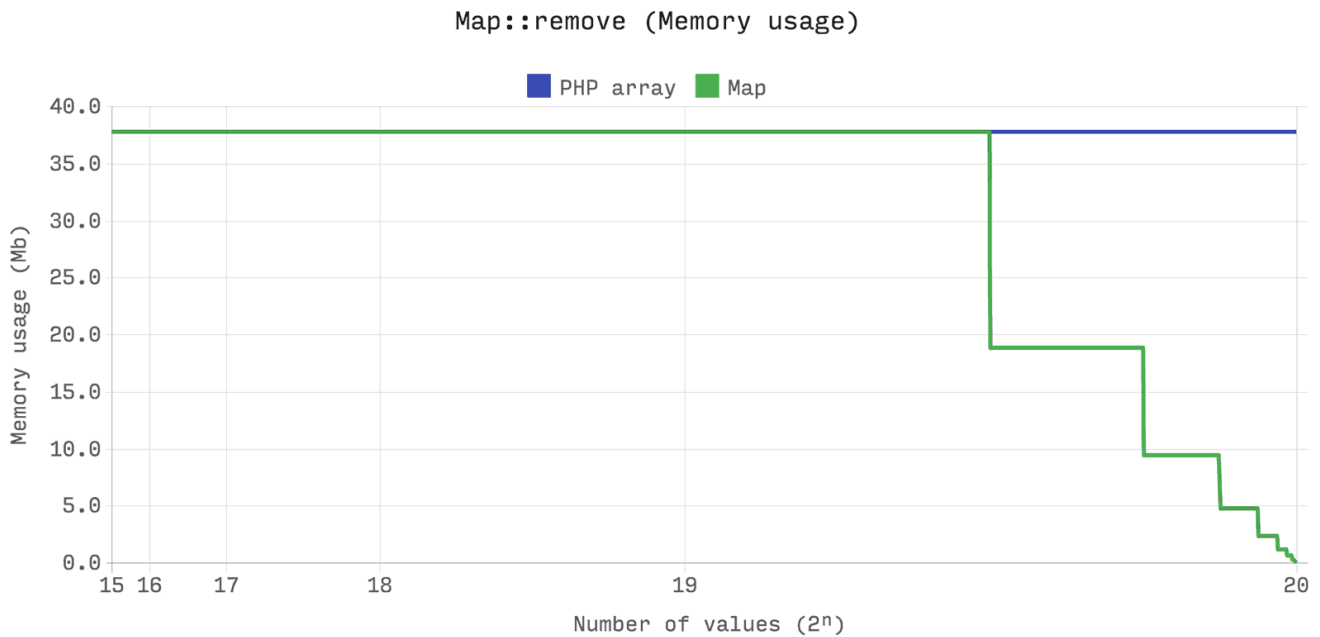
- put, get, remove и containsKey имеют сложность  $O(1)$

## Недостатки

- Не может быть преобразован в array при наличии ключей-объектов
- Нет возможности получить доступ к значениям по индексу (позиции)

Следующий бенчмарк показывает, что производительности и эффективности по памяти между array и Ds\Map идентичны. Однако, array всегда будет держать выделенную память, когда Ds\Map, в свою очередь, освободит память при падении размера ниже четверти своего потенциала.





## Set (Множество)

Set – коллекция **уникальных значений**. Учебники скажут вам, что в структуре Set значения неупорядочены, если реализация не предусматривает иное. Возьмем для примера Java, `java.util.Set` – это интерфейс с двумя основными реализациями: `HashSet` и `TreeSet`. `HashSet` обеспечивает сложность  $O(1)$  для `add` и `remove`, а `TreeSet` обеспечивает сортированный набор данных, но сложность `add` и `remove` возрастает до  $O(\log n)$ .

Set использует ту же внутреннюю структуру, что и Map, также основываясь на array. Это означает, что Set может быть отсортирован за время  $O(n * \log(n))$  когда это понадобится, в остальном он такой же простой как Map и array.

### Сильные стороны

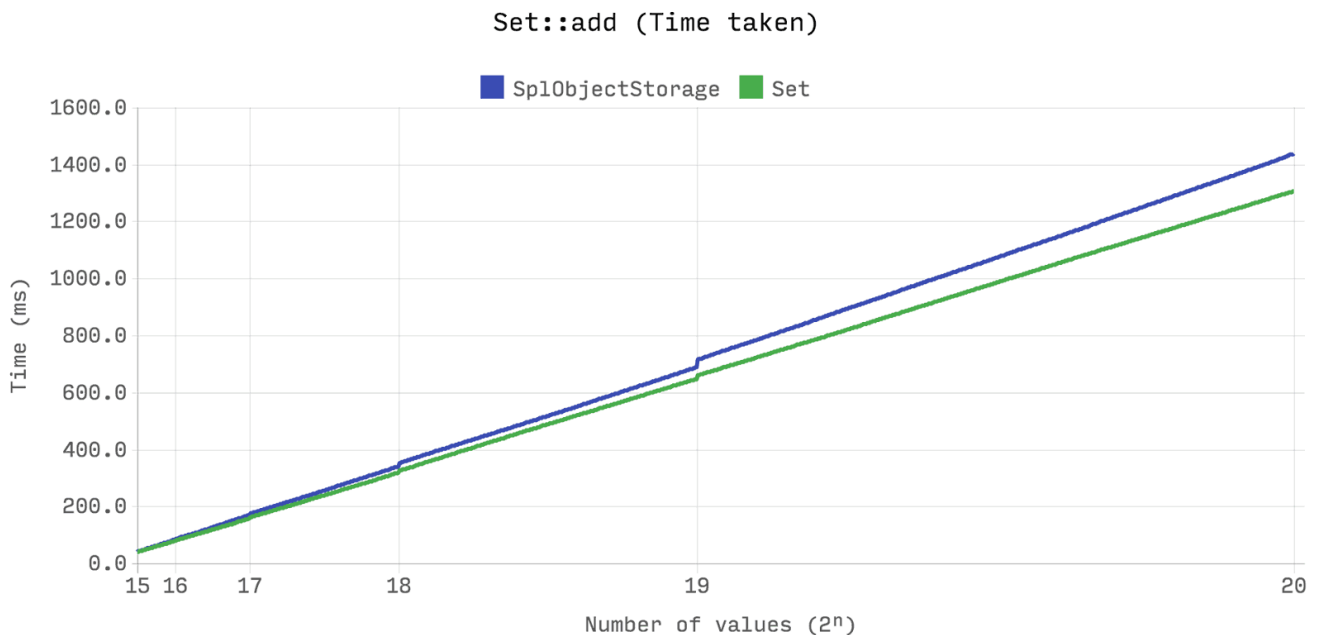
- `add`, `remove` и `contains` имеют сложность  $O(1)$
- Поддерживает работу с объектами, реализующими интерфейс `Hashable`
- Поддерживает **любой тип значений** (`SplObjectStorage` поддерживает только объекты).

- Имеет эквивалент поразрядных логических операций (intersection, difference, union, exclusive or)

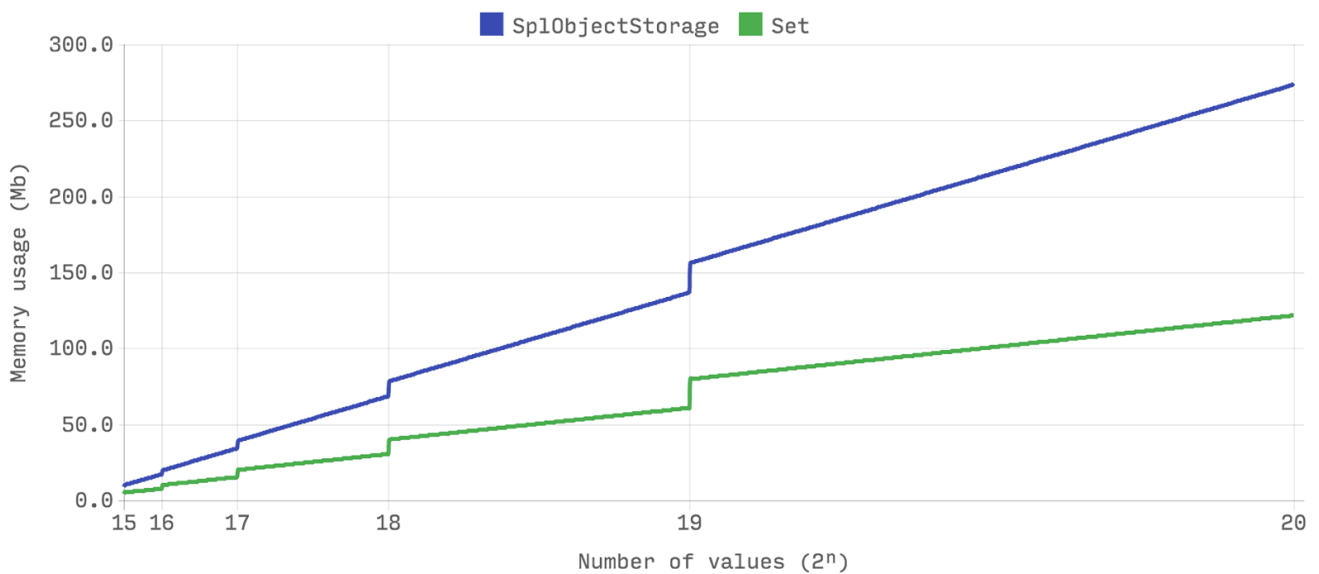
## Недостатки

- Не поддерживает push, pop, insert, shift или unshift
- get имеет сложность  $O(n)$  если есть удаленные значения до момента индексации, в ином случае –  $O(1)$

Следующий бенчмарк показывает время, затраченное на добавление  $2^n$  новых экземпляров stdClass. Он показывает, что Ds\Set **немного быстрее**, чем SplObjectStorage, и использует **примерно в половину меньше** памяти.

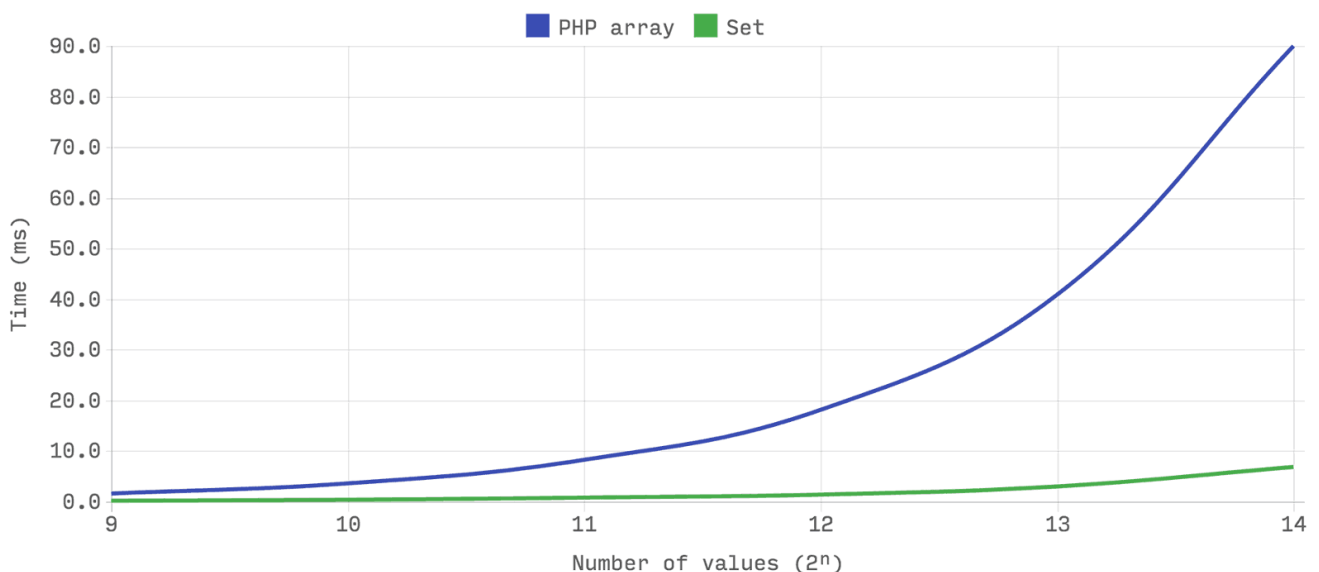


Set::add (Memory usage)



Распространенным способом создания массива с уникальными значениями является `array_unique`, который создает новый массив, содержащий только уникальные значения. Но важно иметь в виду, что **значения в массиве не индексируются**, `in_array` является линейным поиском со сложностью  $O(n)$ . `array_unique` работает только со значениями, без учета ключей, каждая проверка на наличие значения массива – линейный поиск, что дает нам в сумме сложность  $O(n^2)$  по времени и  $O(n)$  по потреблению памяти.

Set vs. array\_unique (Time taken)



# Ответы на ожидаемые вопросы и мнения

## Есть ли тесты?

Сейчас около **2600 тестов**. Вполне возможно, что некоторые тесты являются избыточными, но я предпочел бы косвенно проверить одну и ту же вещь дважды, чем не проверять совсем.

## Документация? Справочник по API?

На момент написания этой статьи пока еще нет полной документации, но она появится вместе с первым стабильным релизом.

Однако, существуют некоторые **хорошо документированные файлы-заглушки**.

## Можем ли мы посмотреть как устроены бенчмарки? Есть что-то о них?

Все бенчмарки прогонялись на стандартном билде PHP 7.0.3 на **2015 Macbook Pro**. Результаты могут отличаться в зависимости от версии и платформы.

## Почему Stack, Queue, Set и Map – не интерфейсы?

Я не верю, что есть необходимость в какой-либо альтернативной реализации. 3 интерфейса и 7 классов – это хороший баланс между прагматизмом и специализацией.

## Когда мне использовать Deque вместо Vector?

Если вы **точно** знаете, что не будете использовать shift и unshift, используйте Vector. Для удобного тайпхинтинга можно указать в качестве типа Sequence.

## Почему все классы являются финализированными?

Дизайн API php-ds применяет парадигму «Composition over



inheritance.

Структуры SPL являются хорошим примером того, как наследование может быть использовано не по назначению. Например, `SplStack` расширяет `SplDoublyLinkedList`, который поддерживает произвольный доступ по индексу, `shift` и `unshift` – так что технически это не стек.

Фреймворк Java-коллекций также имеет несколько интересных случаев, когда наследование порождает двусмысленность. `ArrayDeque` имеет три метода добавления элементов: `add`, `addLast` и `push`. Это не плохо, т.к. `ArrayDeque` имплементирует `Deque` и `Queue`, что объясняет одновременное наличие `addLast` и `push`. Однако, все три метода сразу, делающие одно и то же, вызывают путаницу и непоследовательность.

Старый `java.util.Stack` расширял `java.util.Vector`, тем самым заявляя, что „более полный и последовательный набор операций LIFO обеспечивается интерфейсом `Deque` и его реализациями“, но `Deque` включает в себя методы `addFirst` и `remove(x)`, которые не должны быть частью `stack` структуры по API.

**Просто потому, что эти структуры имеют непересекающиеся методы не значит, что мы не можем так делать.**

На самом деле, это справедливое замечание, но я по-прежнему считаю, что композиция больше подходит для построения структур данных. Они предназначены быть самодостаточными, подобно `array`. Вы не можете отнаследоваться от `array`, он вынуждает вас разрабатывать собственные API вокруг себя, используя его только для хранения фактических данных.

Наследование также вызвало бы лишние сложности во внутренней реализации.

**Зачем нужен еще и `ds` класс в глобальном**

## пространстве имен?

Он обеспечивает альтернативный синтаксис:

```
use Ds\Vector;  
use Ds\Deque;
```

```
$vector = new Vector();  
$deque  = new Deque();
```

```
$vector = ds::vector();  
$deque  = ds::deque();
```

## Почему нет связного списка (*Linked List*)?

Класс `LinkedList` на самом деле появился первым, это казалось хорошим стартом. Но в итоге я решил удалить его, когда понял, что он не сможет конкурировать с `Vector` или `Deque` при любом раскладе. Две основные причины возможной поддержки: **распределение накладных расходов** и **локальность ссылок**.

В связном списке мы добавляем или убираем зарезервированную память для элемента структуры (*node*) всякий раз, когда значение добавляется или удаляется. Нода содержит в себе два указателя (в случае с двусвязным списком), чтобы ссылаться на предыдущую и последующую ноды. Обе структуры, `Vector` и `Deque`, выделяют буфер памяти заранее, поэтому нет необходимости делать это настолько часто. Они также не нуждаются в дополнительных указателях, чтобы знать какое значение до и какое после, тем самым снижаются накладные расходы.

## Будет ли связный список использовать меньше памяти, т.к. там нет буфера?

Только когда коллекция очень мала. Верхней границей количества памяти для `Vector` будет  $(1.5 * (size - 1)) * zval$  байт, не менее  $*10 * zval*$ . В двусвязном списке же будет использоваться  $(size * (zval + 8 + 8))$ . Поэтому связный список будет использовать меньше памяти, чем `Vector` только тогда, когда его размер меньше 6 элементов.

## Окей... связный список использует больше памяти, но почему он медленный?

Узлы *связного списка* обладают плохой **пространственной локальностью**. Это означает, что физическое расположение узла в памяти может быть далеко от прилегающих узлов. Таким образом итерации по связному списку скачут по памяти вместо использования кэша процессора. Значительное преимущество Vector и Deque: элементы физически находятся рядом друг с другом.

*»Несмежность данных в структурах является корнем всех зол производительности. Конкретно, пожалуйста, скажите нет связным спискам«*

*«Нет почти ничего вреднее из того что вы можете сделать чтобы убить все плюсы современных микропроцессоров, чем использовать связный список«*

*– Chandler Carruth (CppCon 2014)*

## PHP – это язык для веб-разработки – производительность не важна.

**Производительность не должна быть вашим главным приоритетом.**

Код должен быть последовательным, ремонтпригодным, надежным, предсказуемым, безопасным и легко понимаемым. Но это не означает, что производительность «не важна».

Мы тратим много времени, пытаюсь уменьшить размер своих ассетов, делаем сравнительный анализ фреймворков и придумываем бессмысленные микро-оптимизации:

- print vs echo, which one is faster?
- The PHP Ternary Operator: Fast or not?
- The PHP Benchmark: setting the record straight
- Disproving the Single Quotes Performance Myth

Но в конечном итоге двухкратный прирост производительности, который приносит с собой PHP7 почему-то всех взбудоражил.

Абсолютно для всех это – одно из главных преимуществ для перехода с PHP5.

Эффективный код позволяет снизить нагрузку на наши сервера. уменьшить время ответа наших API и веб-страниц и снижает время работы наших утилит для разработки. **Высокая производительность важна**, но поддерживаемость кода все же стоит во главе.

---

**Обсуждения:** [Twitter](#), [Reddit](#), [Room 11](#)

**Исходный код:** [github.com/php-ds](https://github.com/php-ds)

**Бенчмарки:** [github.com/php-ds/benchmarks](https://github.com/php-ds/benchmarks)

Автор оригинала: Rudi Theunissen