The Auckland PHP Meetup Group

Data Structures in PHP 7

Rudi Theunissen

Developer at Figured

rtheunissen@php.net
github.com/rtheunissen

Arrays in PHP

- Complex, flexible, master-of-none, hybrid structure.
- It’s like a Javascript array and object combined.
- **Pragmatic**: *“practical”,* rather than *“theoretical”.*
- Optimised for everything, optimised for nothing.
- Used internally for object properties.

How do PHP arrays work?

Let’s do a quick recap of a *hash table…*

Hash Tables

- A structure used to associate a **key** with a **value**.
- Can usually find a key’s associated value very quickly.
- Computers can only use integers for reference, but humans often want to associate other things, like characters or objects.
- For example, a dictionary associates a word (key) with its definition (value). We can find the definition quickly because the words (keys) are in alphabetical order.
- A **hashing function** is used to translate keys to integers.

Basic Hash Table

- Usually has an internal **buffer** of **buckets**.
- A **bucket** is a container for a *key* and *value*.
- The **buffer** is an allocated block of memory to store the buckets, and has a **capacity**.
- Does not allow duplicate keys.
- Let’s take a look at a simple example…

Linked Lists

- The hash table example uses a *linked list* to chain the buckets together.

- Unfortunately, linked lists are **bad**.
  They are slow and use a lot of memory.

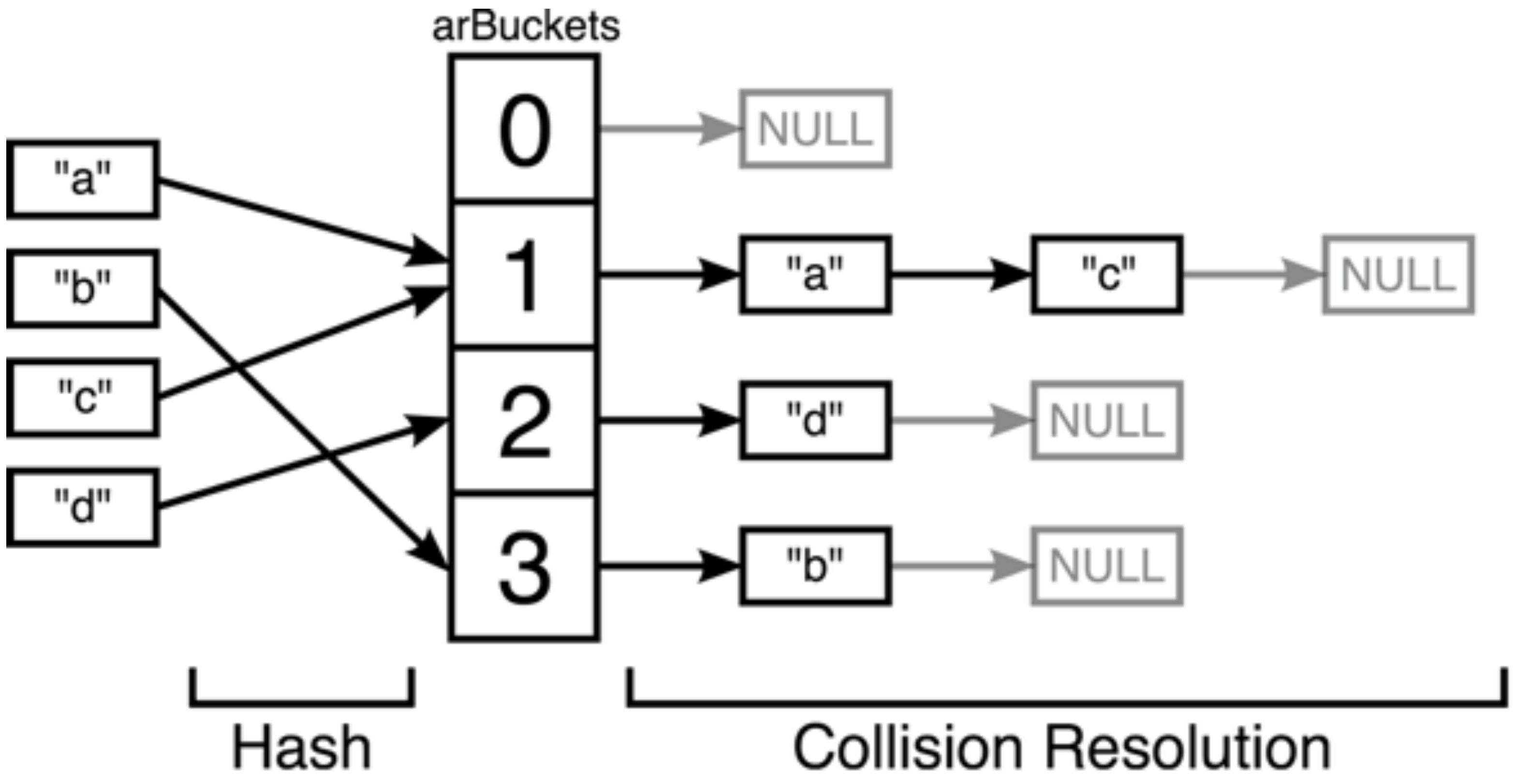
- Memory allocation is expensive, and we have to allocate each bucket individually.

- The buckets are allocated all over the place in memory, which means we don’t have *spatial locality.*

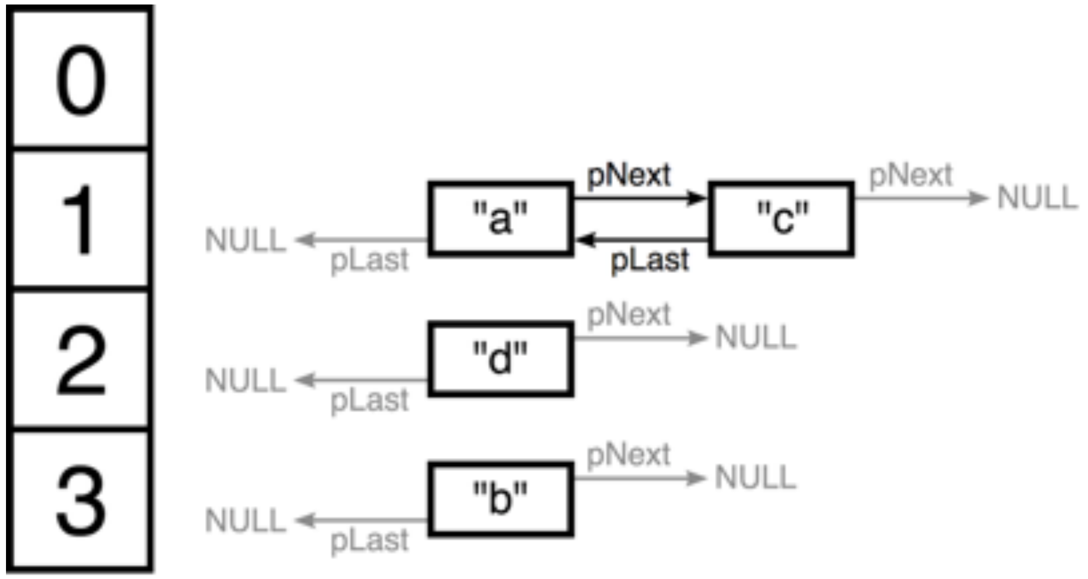
- We want to minimise allocations and maximise locality.

Arrays in PHP 5

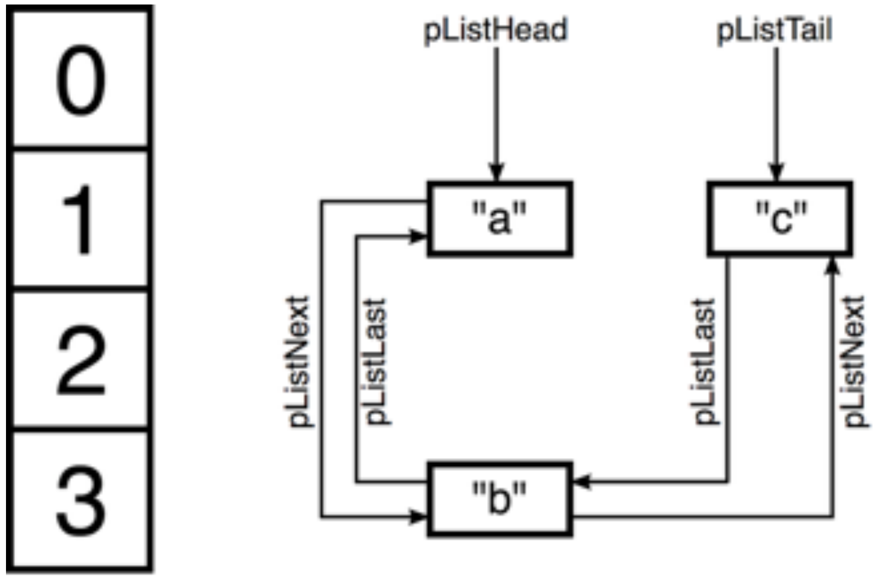
- Use a doubly-linked list for the collision chain.
- Use a doubly-linked list to maintain insertion order.
- Have been rewritten completely in PHP 7.
- Require about **3.5x more memory** than in PHP 7.
- Significantly slower than arrays in PHP 7 because of cache-unfriendly traversal (no spatial locality) and many allocations.



arBuckets
0
1
2
3
"a"
"b"
"c"
"d"
NULL
"a"
"c"
NULL
"d"
NULL
"b"
NULL
Hash
Collision Resolution



0
1
2
3
"a"
"c"
"d"
"b"
pNext
pLast
NULL

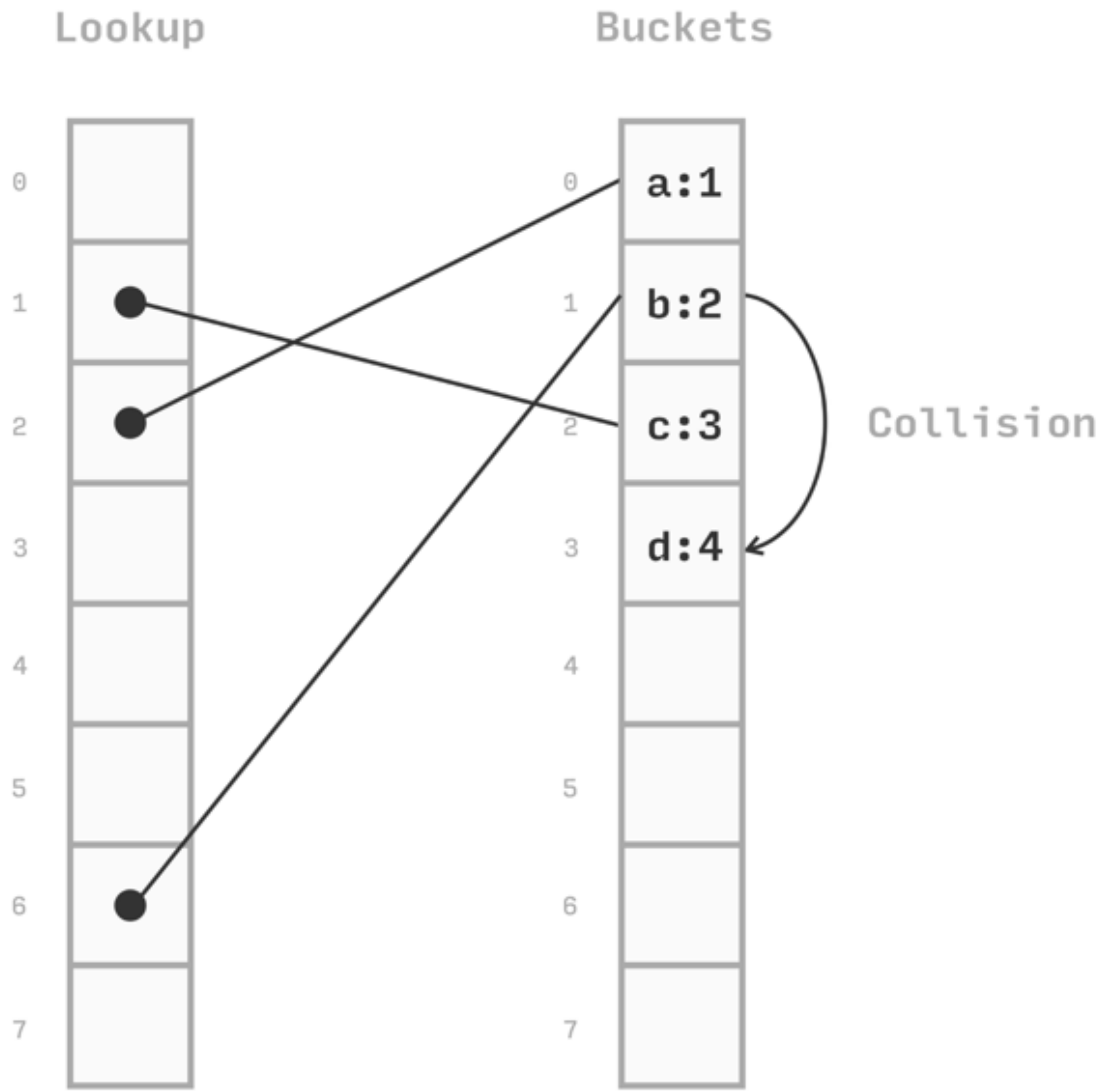


0
1
2
3
pListHead
pListTail
"a"
"c"
"b"
pListNext
pListLast

Arrays in PHP 7

- ✅ Can allocate many buckets at once.
- ✅ Separate the bucket buffer from the hash lookup.
- ✅ Don’t use linked lists at all.
- ❌ Still only support scalar keys.
- ❌ Still have the same limitations, because it’s still a hybrid structure that attempts to do everything.

```
$array["a"] = 1;
$array["b"] = 2;
$array["c"] = 3;
$array["d"] = 4;

hash("a"); // 2
hash("b"); // 6
hash("c"); // 1
hash("d"); // 14

// 14 % $capacity = 6
```



Can we do better?

- Arrays get the job done, but can be **overkill**.
  We don’t need a hash table for a basic list of elements.
- You can’t know how an array has been used without inspecting its contents. Is it associative (uses keys) or just a list of elements? ¯_(ツ)_/¯
- Keys can only be **strings** or **integers**, and numeric string keys automatically become integers.
- Other programming languages have **collections**, or at least distinguish between a list and a dictionary, like `[]` and `{}` in Javascript and Python.

Collections

- *"Collections"* is often used to describe a library or selection of structures.
- Each structure is optimised for different use cases, so you get to pick which one you want to use based on what task you have at hand.
- Most major languages have collections.
    - Java has the *Java Collections Framework*
    - Javascript has *arrays* and *objects*, but also new ones like *Map* and *Set*.
    - Python has a *list*, *dict*, *tuple*, *set* and *deque (a few others too).*
    - Hack has native data structures like *Map*, *Set*, and *Vector*.
    - Ruby, C#, C++…

What about PHP?

- PHP has what’s called the *SPL Data Structures*.
- They were written around 2009 for PHP 5.3.
- They are poorly designed and don’t offer any performance benefits.
- Since PHP 7 they are now significantly slower and use more memory than arrays.
- Don’t use them.

Poorly designed?

- Let’s take a look at SplDoublyLinkedList
- A linked list with both “previous” and “next” links.
- Linked lists are bad.
- SplStack and SplQueue extend it, so they both inherit the linked list methods and behaviour.
- SplObjectStorage is both a *Map* and *Set*, but keys *have to be objects.* ¯\_(ツ)_/¯

- SplObjectStorage doesn't work as expected when you *foreach* through it.

- SplObjectStorage has a *getHash* function, which means the structure determines the object's integer representation rather than the object.

- There isn't a nice way to merge an SplObjectStorage instance with another one.

- I've never seen anyone use any of these structures.

- PHP deserves better than this. 🐘

php-ds

- A data structure extension written in C for PHP 7
- Provides specialised, efficient data structures as alternatives to the standard array.
- Stable, tested, documented.
- There is a PHP implementation that can be installed with Composer that acts as a fallback for when the extension isn't installed.
- An attempt to replace the SPL data structures.

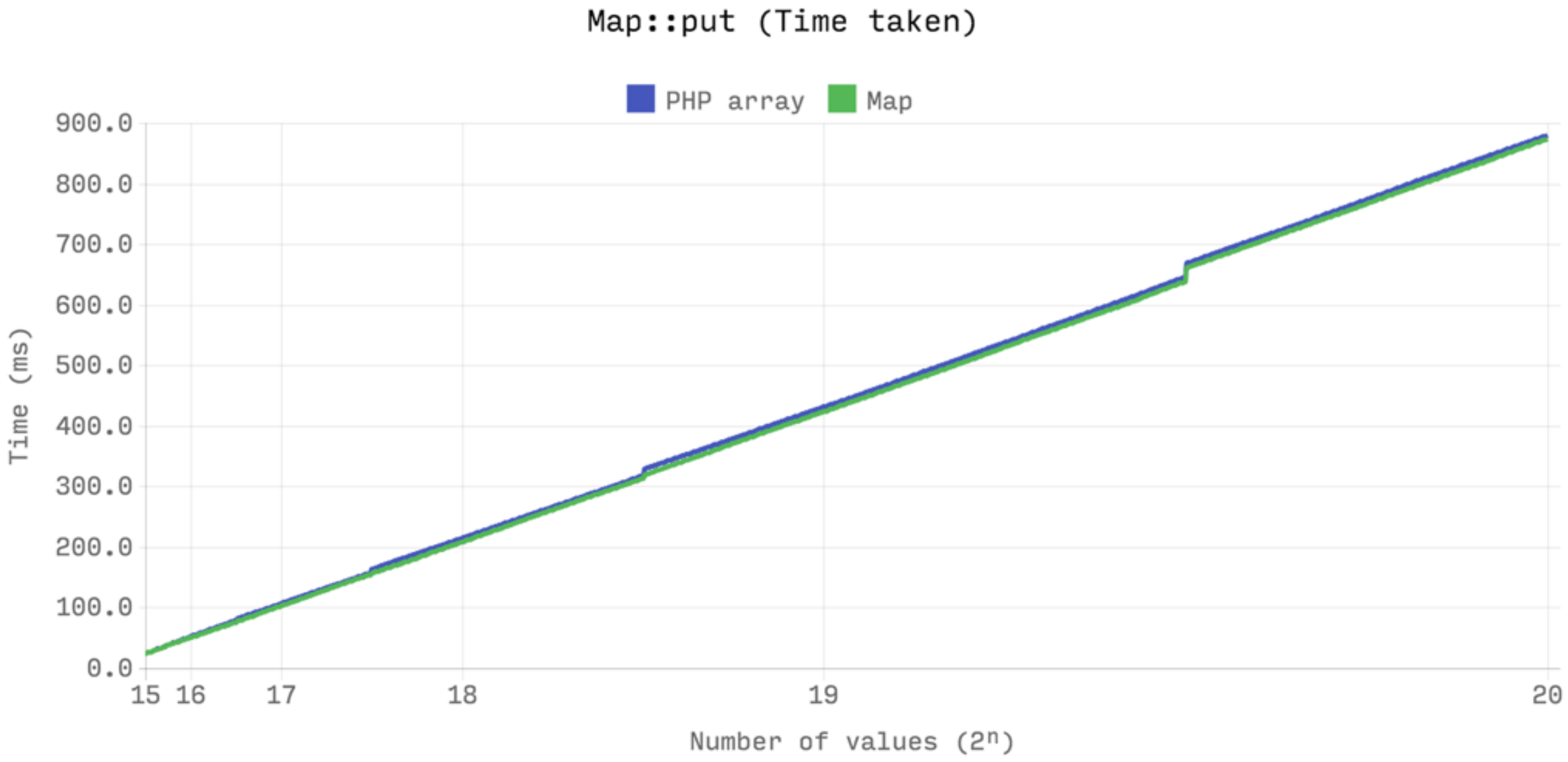
```
interface **Hashable**

interface **Collection**

interface **Sequence** extends Collection

final class **Vector** implements Sequence

final class **Deque** implements Sequence

final class **Map** implements Collection

final class **Set** implements Collection

final class **Stack** implements Collection

final class **Queue** implements Collection

final class **PriorityQueue** implements Collection
```

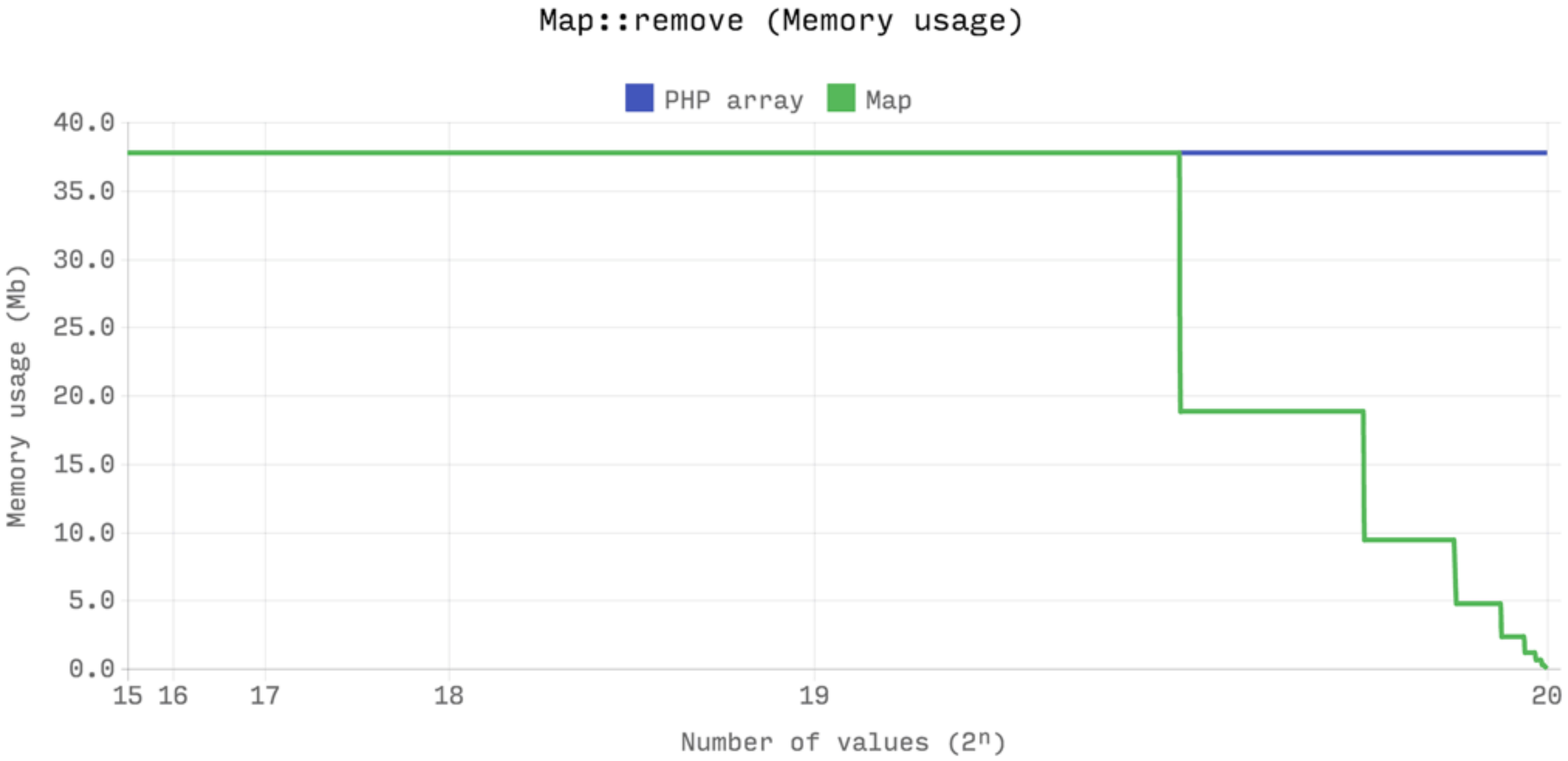
Hashable

- Allows objects to define their own hash function.
- Provides two methods: **hash** and **equals**.
- Honoured by **Map** and **Set**.
- If an object doesn’t implement *Hashable*, the default hash function is *spl_object_hash.*
- An object’s hash value should never change,
  but doesn’t have to be unique.
- For example, a *Person* object might return their birth date.

Map

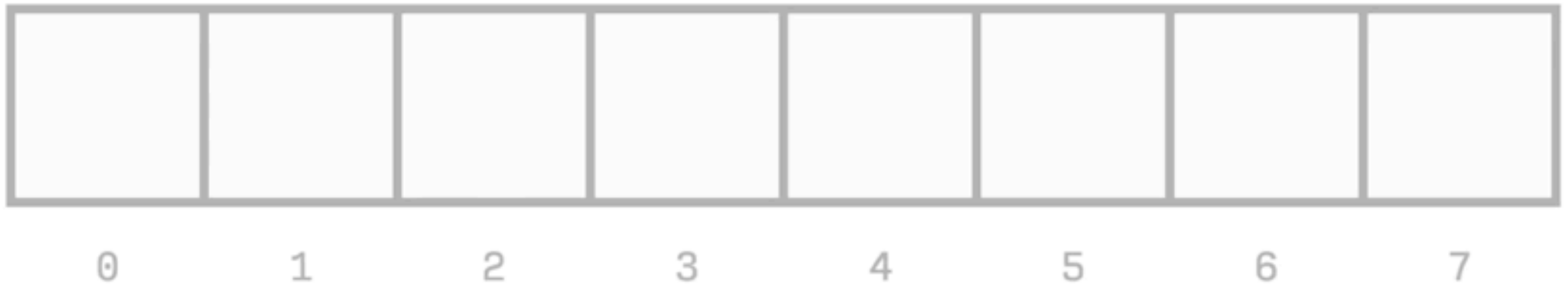
- Keys can be any type, including objects.
- Honours the *Hashable* interface.
- Performance and memory use is effectively identical to an array.
- Insertion order is preserved.
- Supports array syntax.
- *foreach* works as expected.
- Automatically releases memory (reduces the size of the buffer) when the number of elements drop below a threshold.


Map::put (Time taken)
PHP array
Map
Time (ms)
900.0
800.0
700.0
600.0
500.0
400.0
300.0
200.0
100.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)

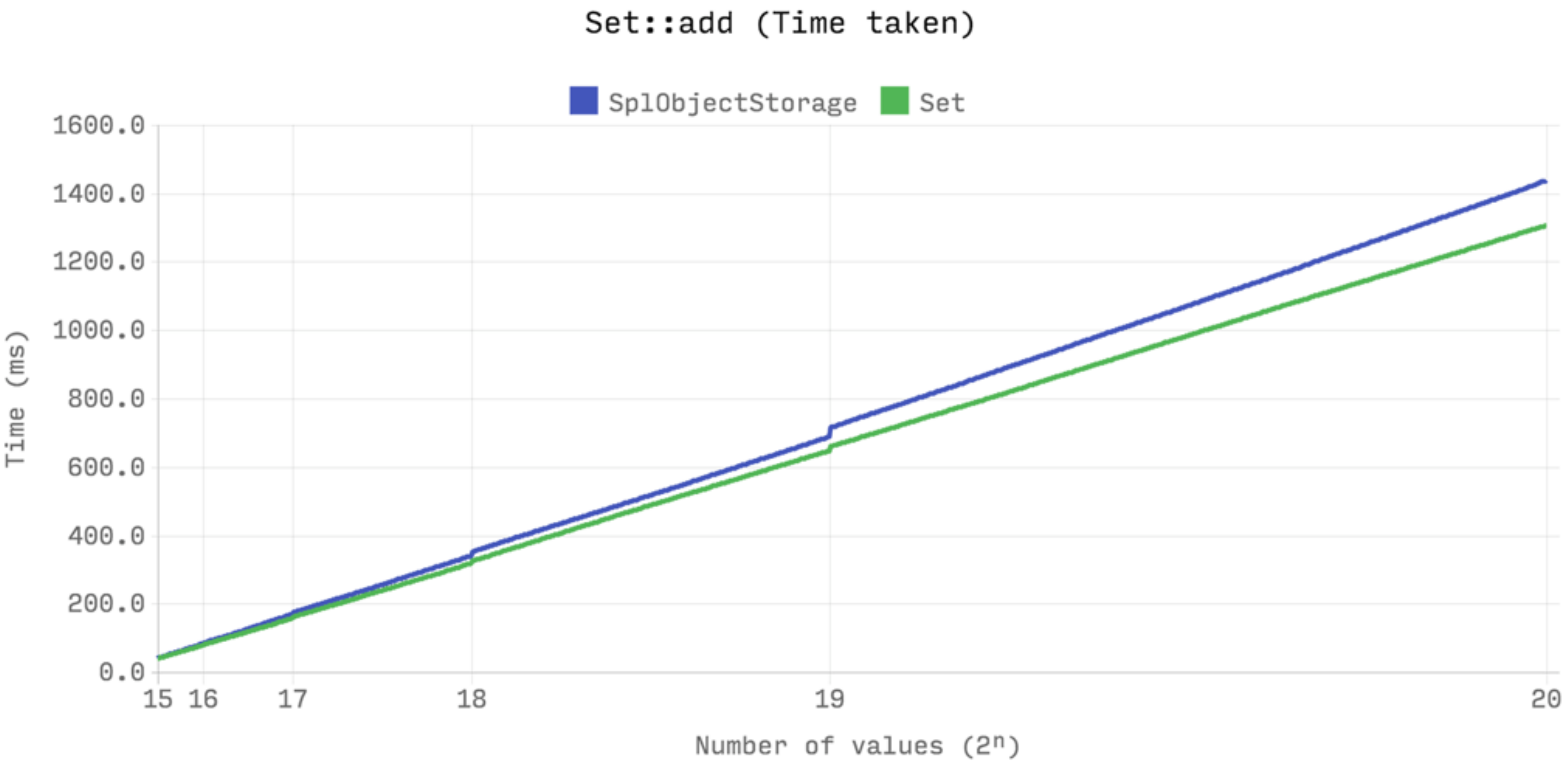

Map::remove (Memory usage)
PHP array
Map
40.0
35.0
30.0
25.0
20.0
15.0
10.0
5.0
0.0
Memory usage (Mb)
15
16
17
18
19
20
Number of values (2ⁿ)

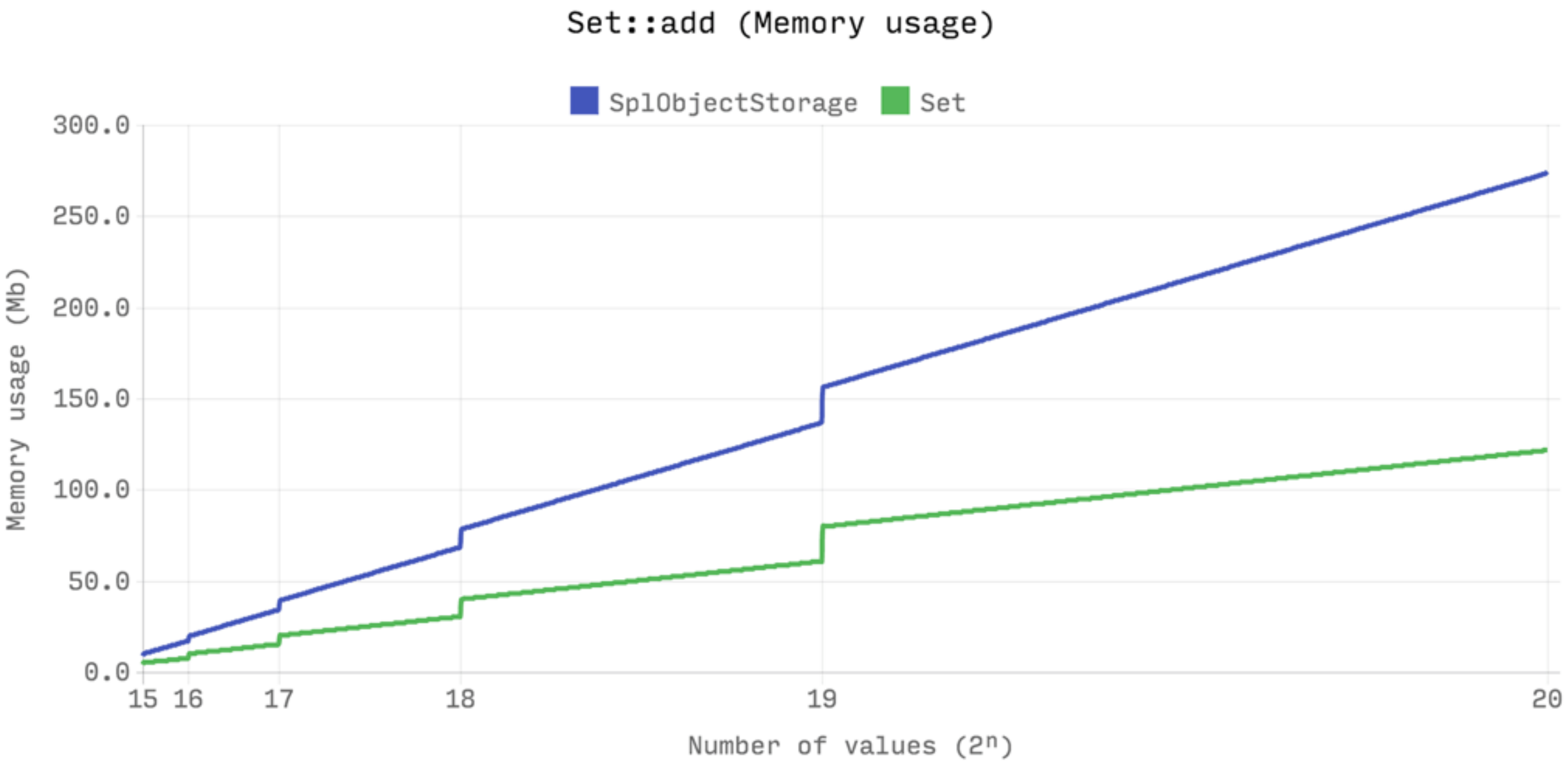
Set

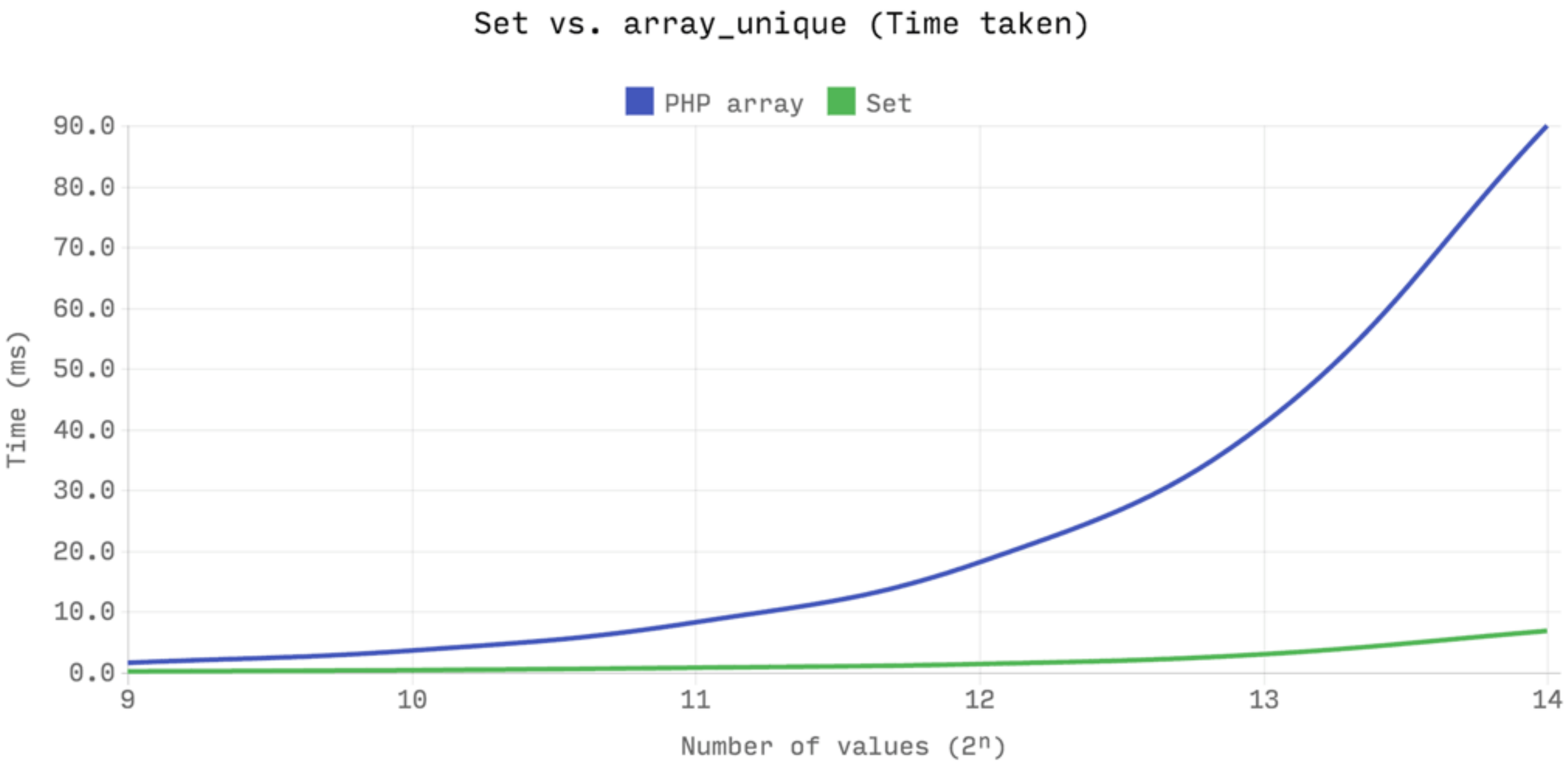
- A collection of **unique values**.
- Attempting to add the same value more than once will do nothing, so uniqueness is enforced.
- Very fast to add, delete, or check if a value exists in the set.
- Values can be of any type.
- Uses the same internal structure as *Map,* which means that performance is the same as an array.
- Insertion order is preserved.
- Supports array syntax.



```
$set->add('A');
$set->add('B');
$set->add('C');
```

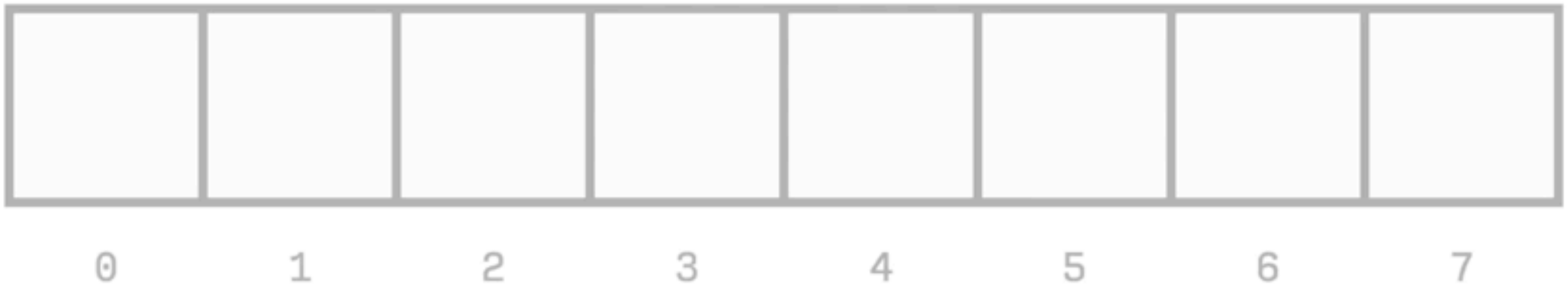

Set::add (Time taken)
SplObjectStorage
Set
Time (ms)
1600.0
1400.0
1200.0
1000.0
800.0
600.0
400.0
200.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)


Set::add (Memory usage)
SplObjectStorage
Set
Memory usage (Mb)
300.0
250.0
200.0
150.0
100.0
50.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)

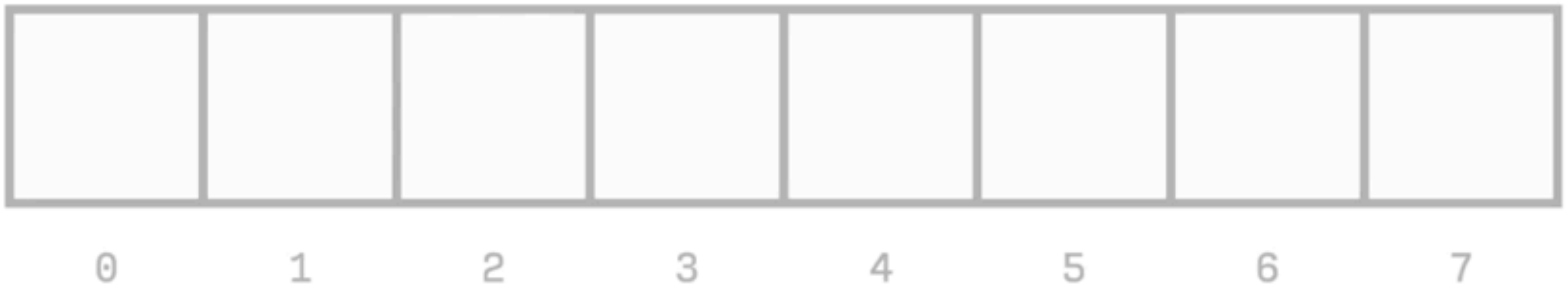

Set vs. array_unique (Time taken)
🟦 PHP array
🟩 Set
Time (ms)
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
9
10
11
12
13
14
Number of values (2^n)

Sequence

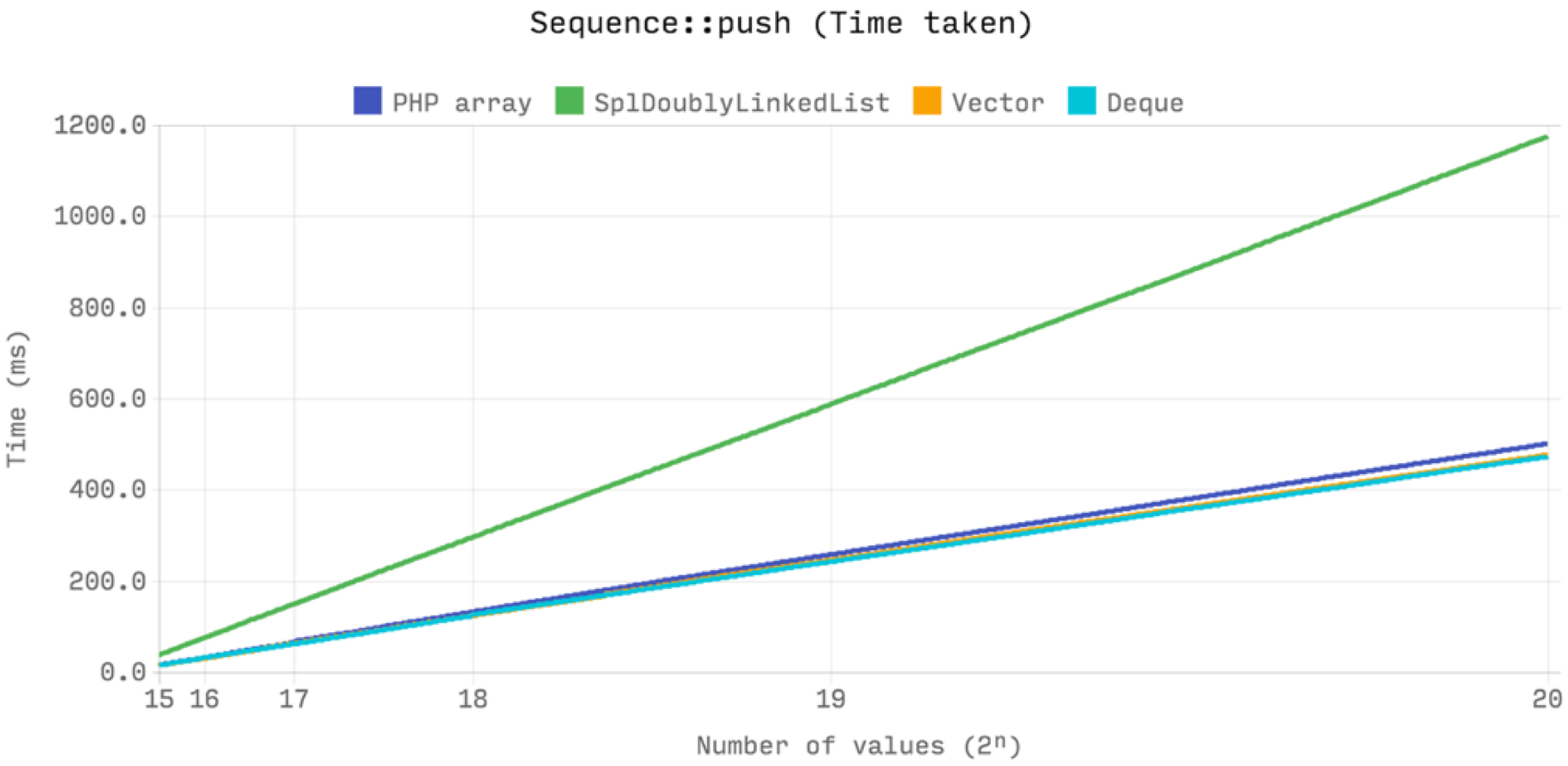
- Two implementations: **Vector** and **Deque**.
- Doesn’t use keys at all.
- Offsets are integers between 0 and (*size* - 1).
- Very similar to a Javascript array.
- Doesn’t use a hash table internally so there are no buckets or linked lists, which allows a sequence to use less memory than arrays, maps, and sets.
- Vector and Deque will be merged into Sequence in v2.0

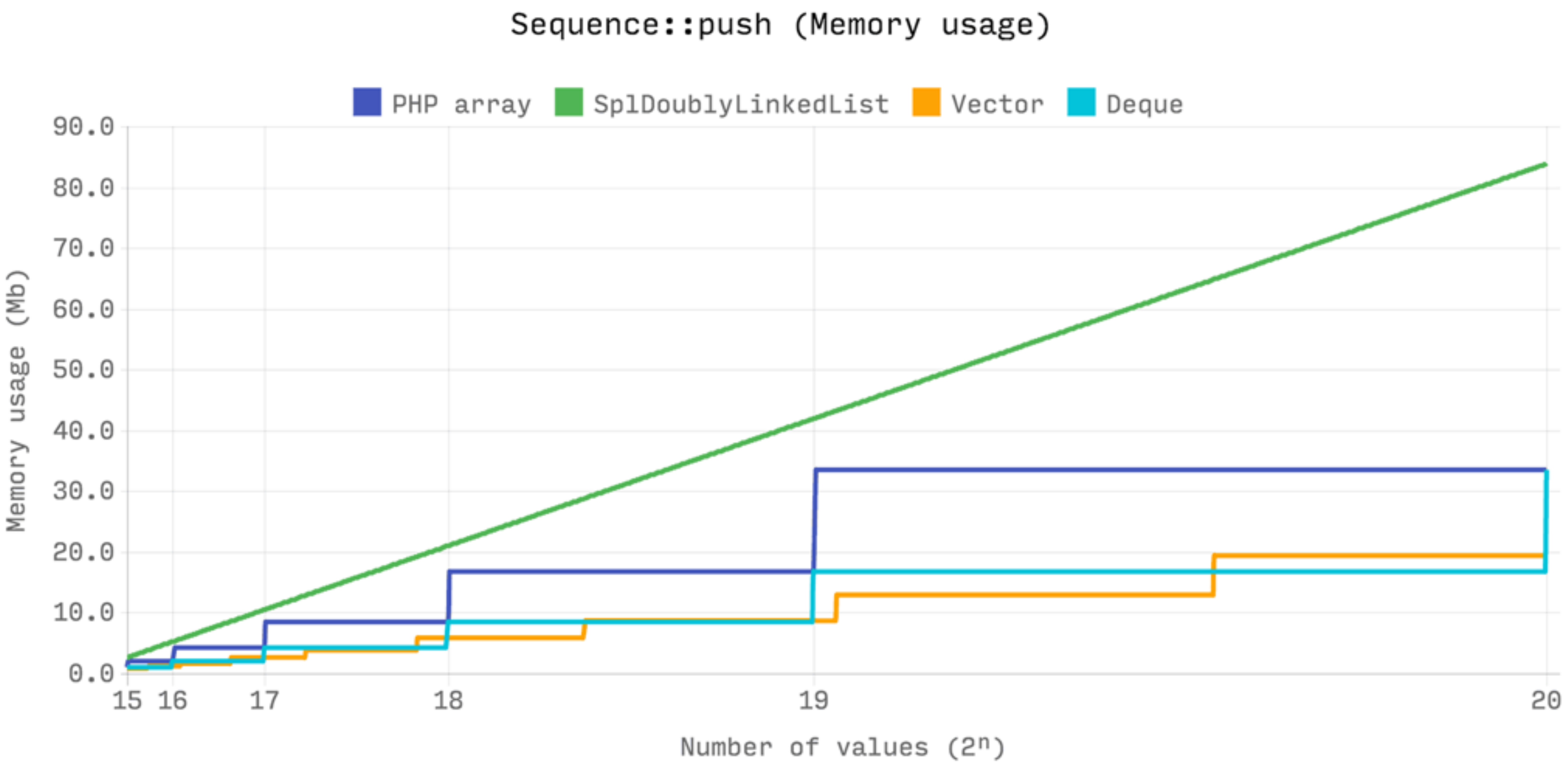


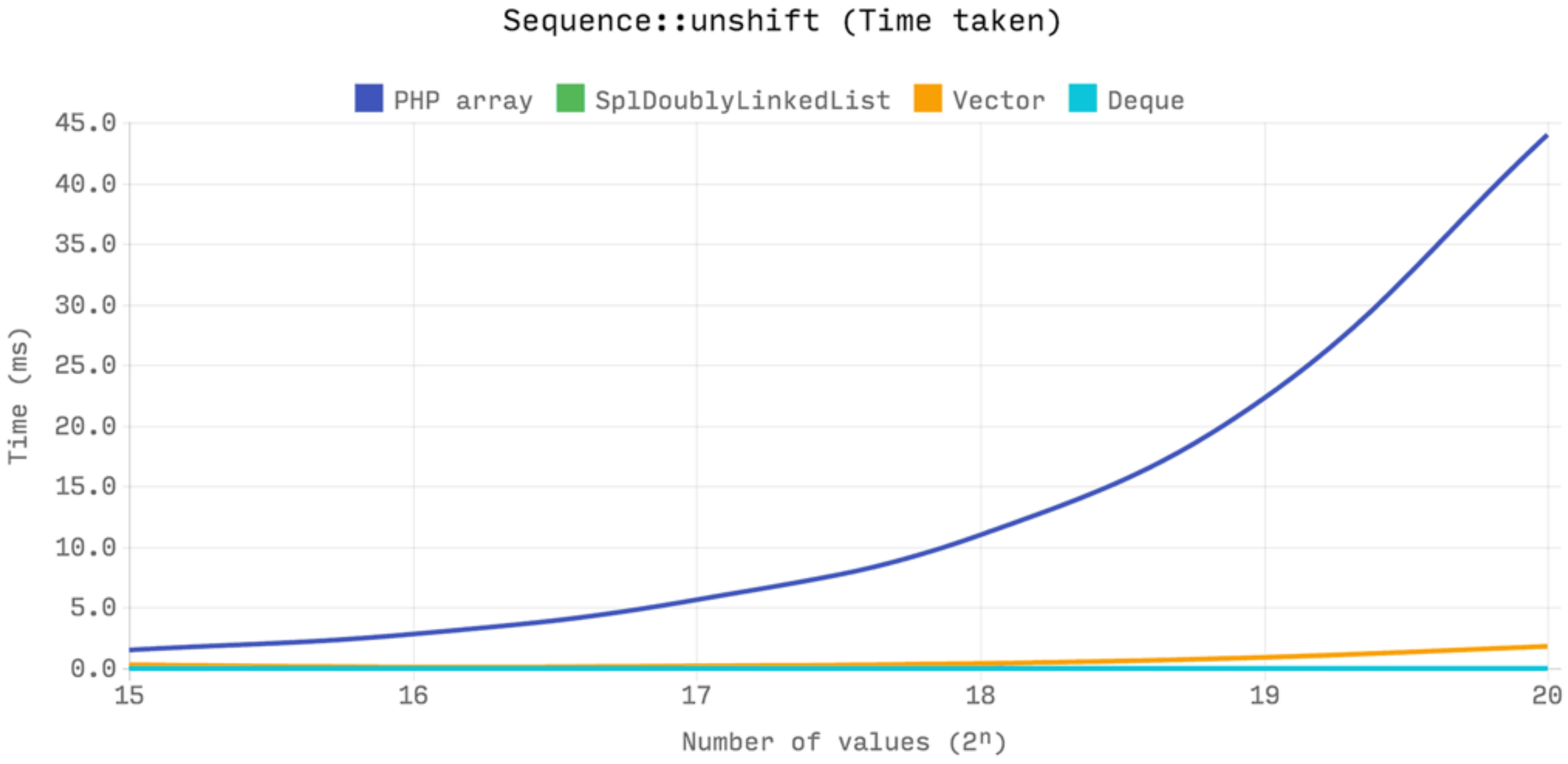
```
$vector->push('D');
$vector->push('E');
$vector->push('F');
```

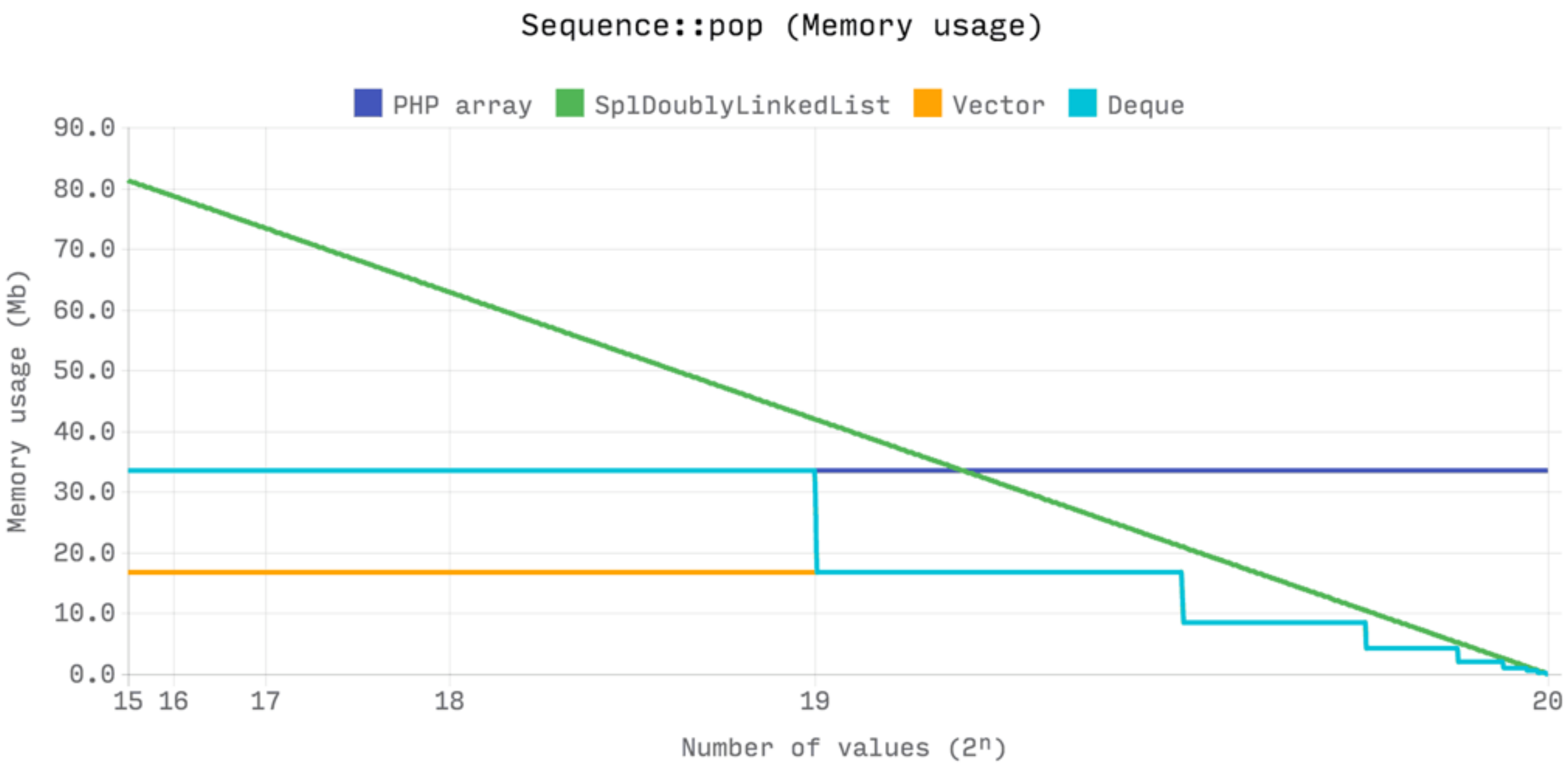


```
$deque->push('D');
$deque->push('E');
$deque->push('F');
```


Sequence::push (Time taken)
PHP array
SplDoublyLinkedList
Vector
Deque
1200.0
1000.0
800.0
600.0
400.0
200.0
0.0
Time (ms)
15
16
17
18
19
20
Number of values (2ⁿ)


Sequence::push (Memory usage)
PHP array
SplDoublyLinkedList
Vector
Deque
Memory usage (Mb)
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)


Sequence::unshift (Time taken)
PHP array
SplDoublyLinkedList
Vector
Deque
Time (ms)
45.0
40.0
35.0
30.0
25.0
20.0
15.0
10.0
5.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)

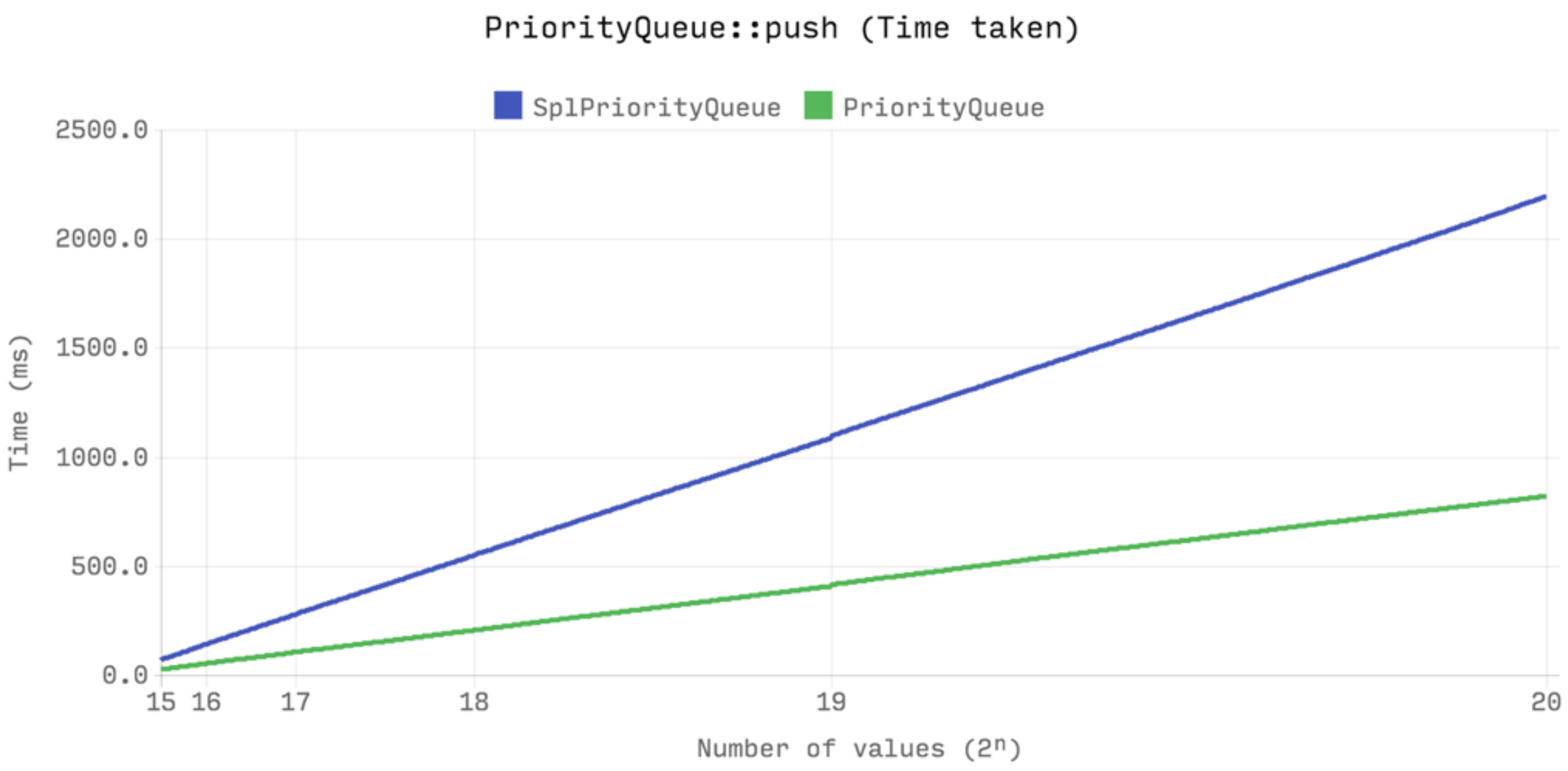

Sequence::pop (Memory usage)
PHP array
SplDoublyLinkedList
Vector
Deque
Memory usage (Mb)
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)

Stack and Queue

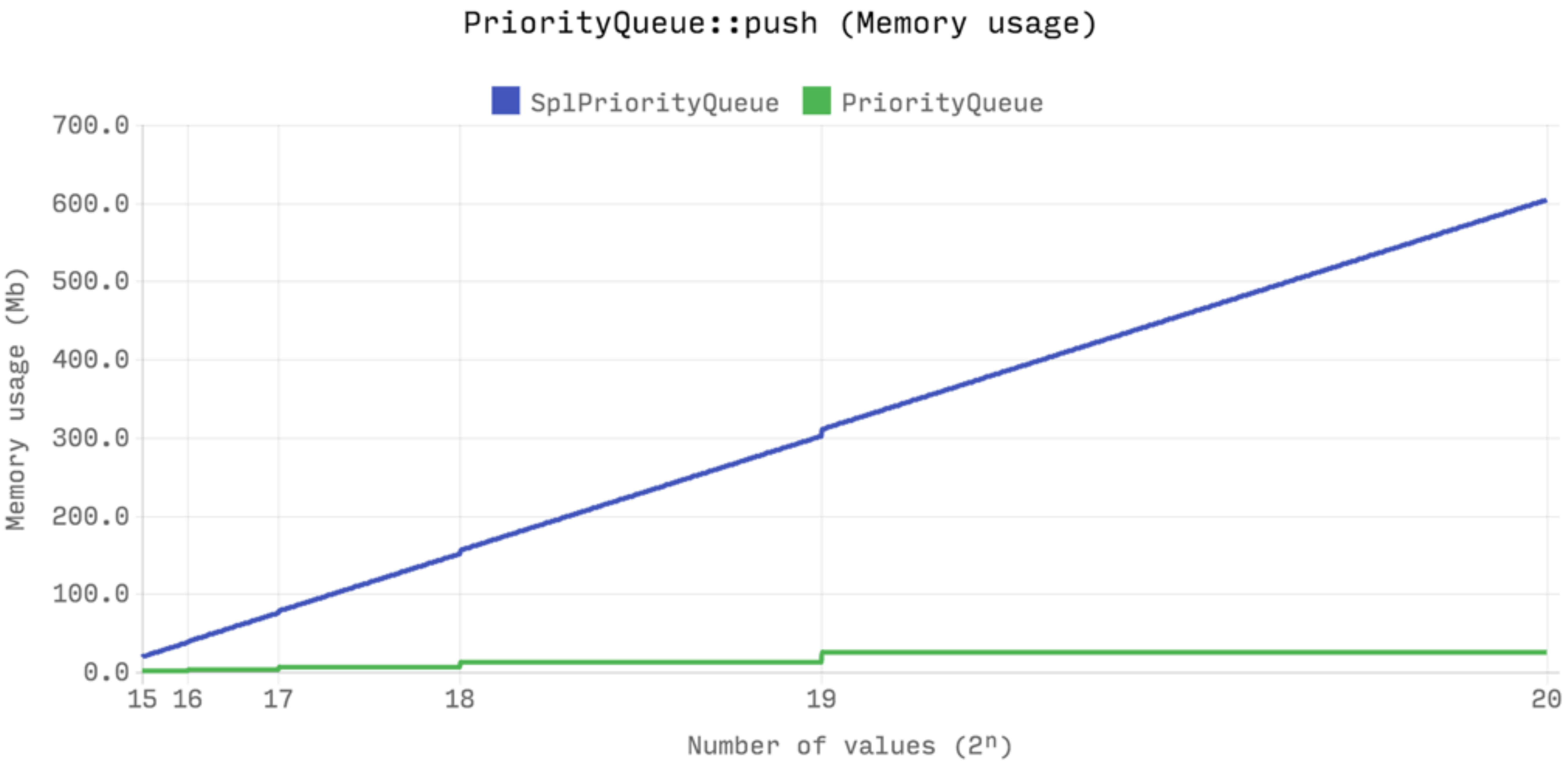
- **Stack** uses a Vector internally.
- **Queue** uses a Deque internally.
- Iterates destructively.
- Can’t access elements in the middle of the sequence, only at one end.
- Stack is *first-in-last-out*, like a stack of pancakes.
- Queue is first-in-first-out, like a queue at the post office.

PriorityQueue

- Very similar to a Queue.
- Values are added along with a *priority*.
- The value with the highest priority will be at the front of the queue.
- Values with the same priority fall back to the behaviour of a queue: *first-in-first-out*.
- More than **twice as fast** as SplPriorityQueue.
- Uses **20 times less memory** than SplPriorityQueue.



PriorityQueue::push (Time taken)
SplPriorityQueue
PriorityQueue
2500.0
2000.0
1500.0
1000.0
500.0
0.0
Time (ms)
15
16
17
18
19
20
Number of values (2ⁿ)


PriorityQueue::push (Memory usage)
SplPriorityQueue
PriorityQueue
700.0
600.0
500.0
400.0
300.0
200.0
100.0
0.0
Memory usage (Mb)
15
16
17
18
19
20
Number of values (2ⁿ)

Wait… 20 times?

- The reason why SplPriorityQueue uses that much time and memory is because it **uses a PHP array** for each value/priority pair.

- The catch is that a PHP array has a minimum capacity of 8 buckets. So each value that you add to an SplPriorityQueue will allocate 8 buckets but only use 2, when it actually only needs 1.

- ¯_(ツ)_/¯

Should we be using php-ds?

- It’s still too early for real change in the ecosystem.
- Hopefully *php-ds* can be a default extension in PHP 8.
- We lose all the performance benefits if we still use arrays between the database and the collections.
- Don’t forget about the semantic benefits.
- So **yes**, at least instead of the SPL structures, and for sure if and when it becomes a default extension.

Any questions or comments?

github.com/php-ds