[Arrays]

Rudi Theunissen
rtheunissen@php.net

Recap: *Complexity*

How does the size of a dataset influence performance?

"Big-Oh notation" is the worst-case amount of work required.

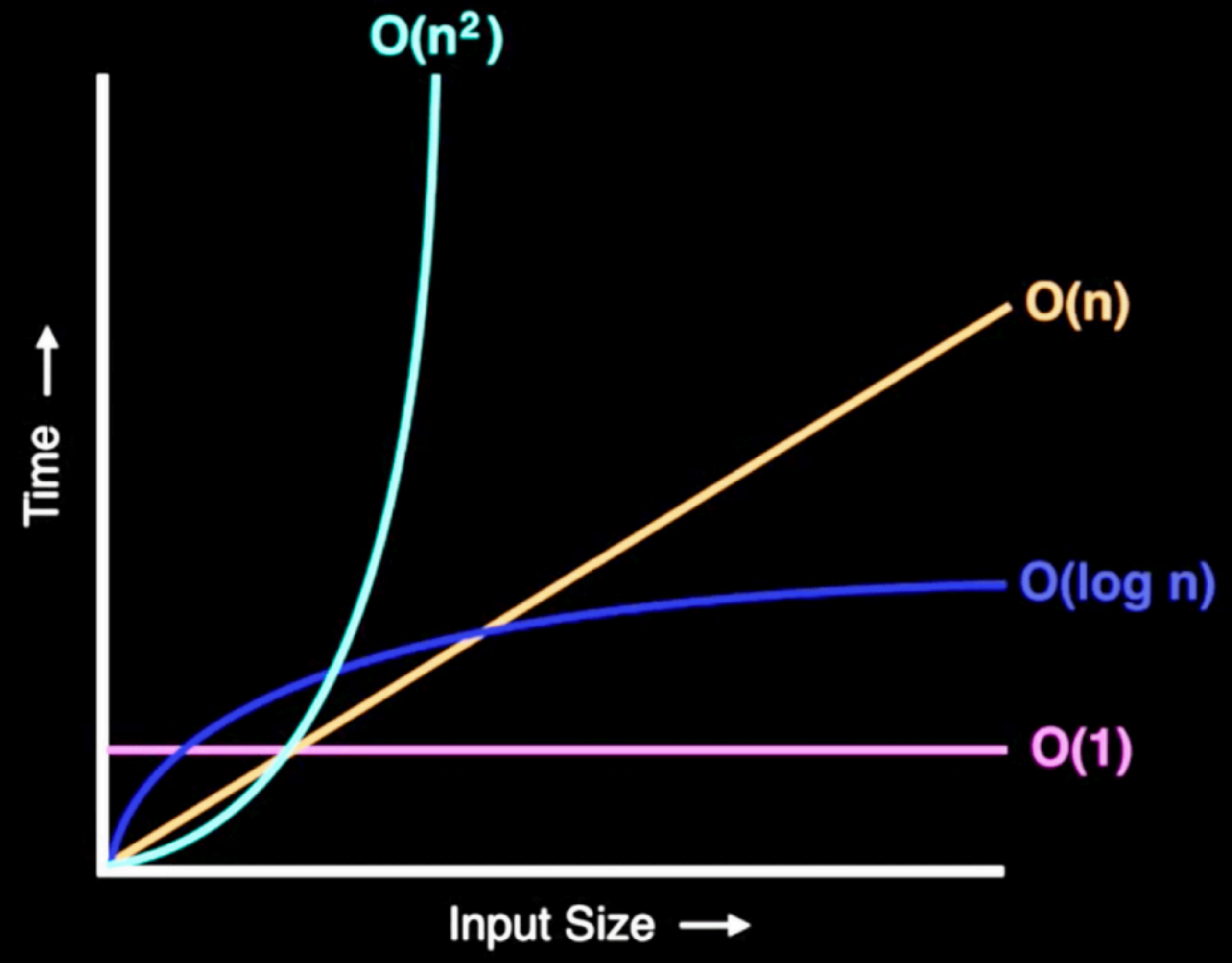
Applies to both ***time*** (performance) and ***space*** (efficiency).

"Amortized" means averaged over many operations.

We have to balance simplicity, performance, and efficiency.

"Big-Oh notation" does not always translate to practical results.

It is a good predictor for what we can expect in practice.



O(n²)	*Exponential*	Very bad
O(n)	*Linear*	Bad
O(log n)	*Logarithmic*	Good
O(1)	*Constant*	Very good

Recap: *Ordering*

Sequential	By position, one after another.
Sorted	By value, usually comparison-based.
Unordered	Arbitrary, unreliable.

Recap: *Data Structures*

List Linear, sequential.

Set Linear, distinct values, ordered or unordered.

Map Associative, distinct keys, ordered or unordered.

Recap: *Data Structures*

List Linear, sequential.

Set Linear, distinct values, ordered or unordered.

Map Associative, distinct keys, ordered or unordered.

Arrays: *Flexibility*

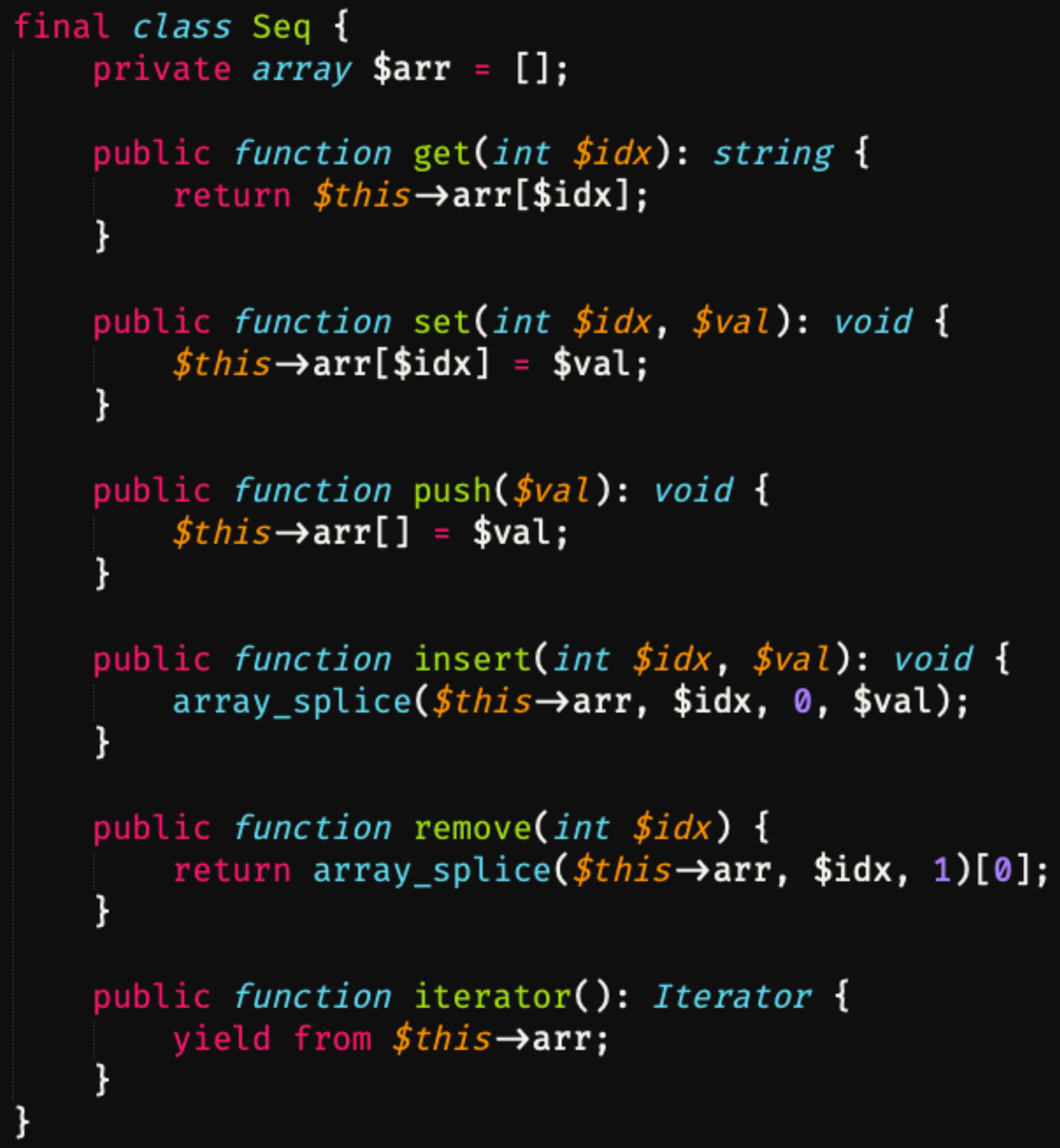
Arrays are linear, associative, and sequential (insertion order).

Similar in behavior to:
  Java: *LinkedHashMap* combined with *ArrayList*
  Javascript: *Map* combined with *Array*

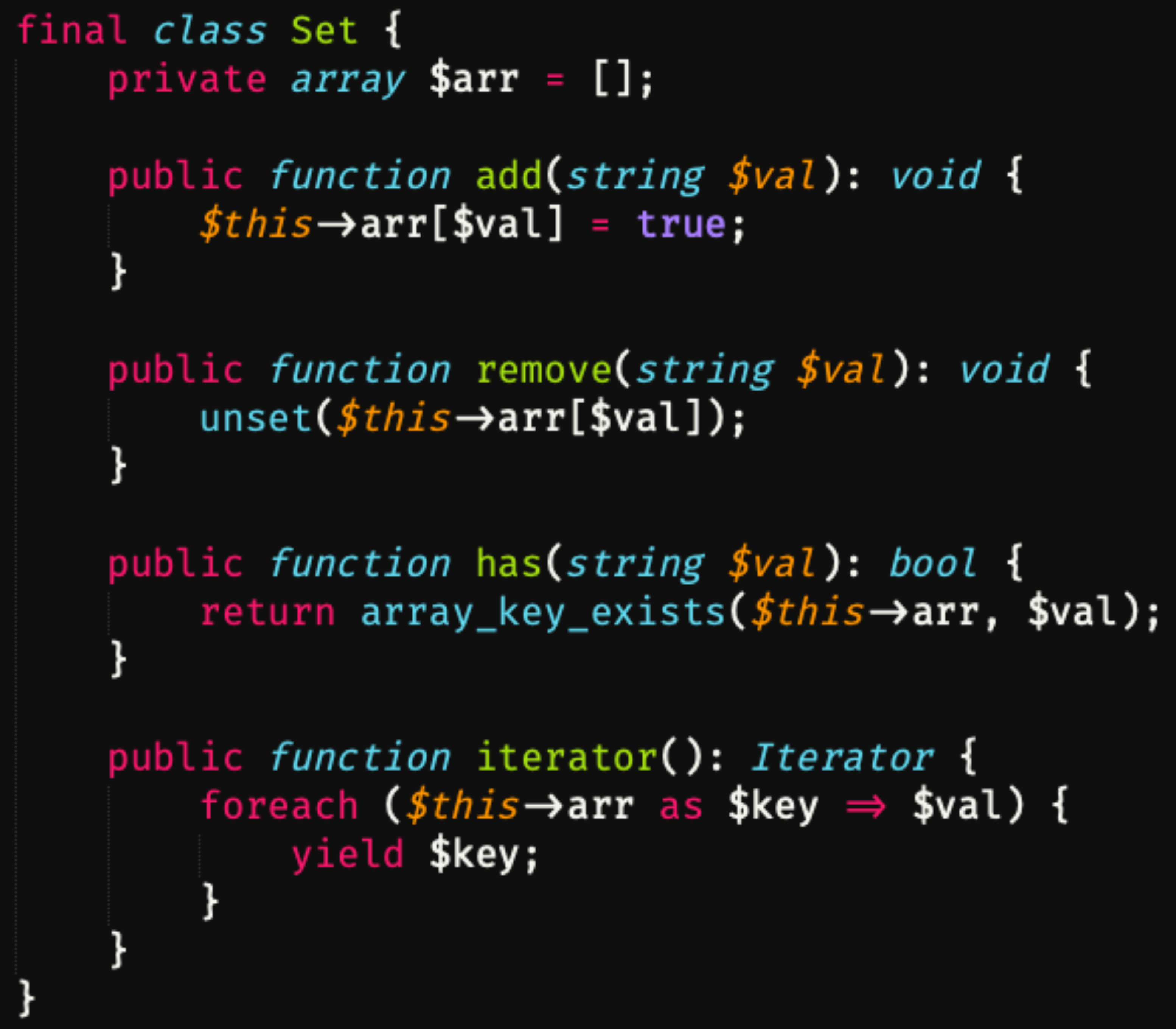
We can use *array* for everything!

No need to know a lot about data structures.
No need to consider when to use which one.

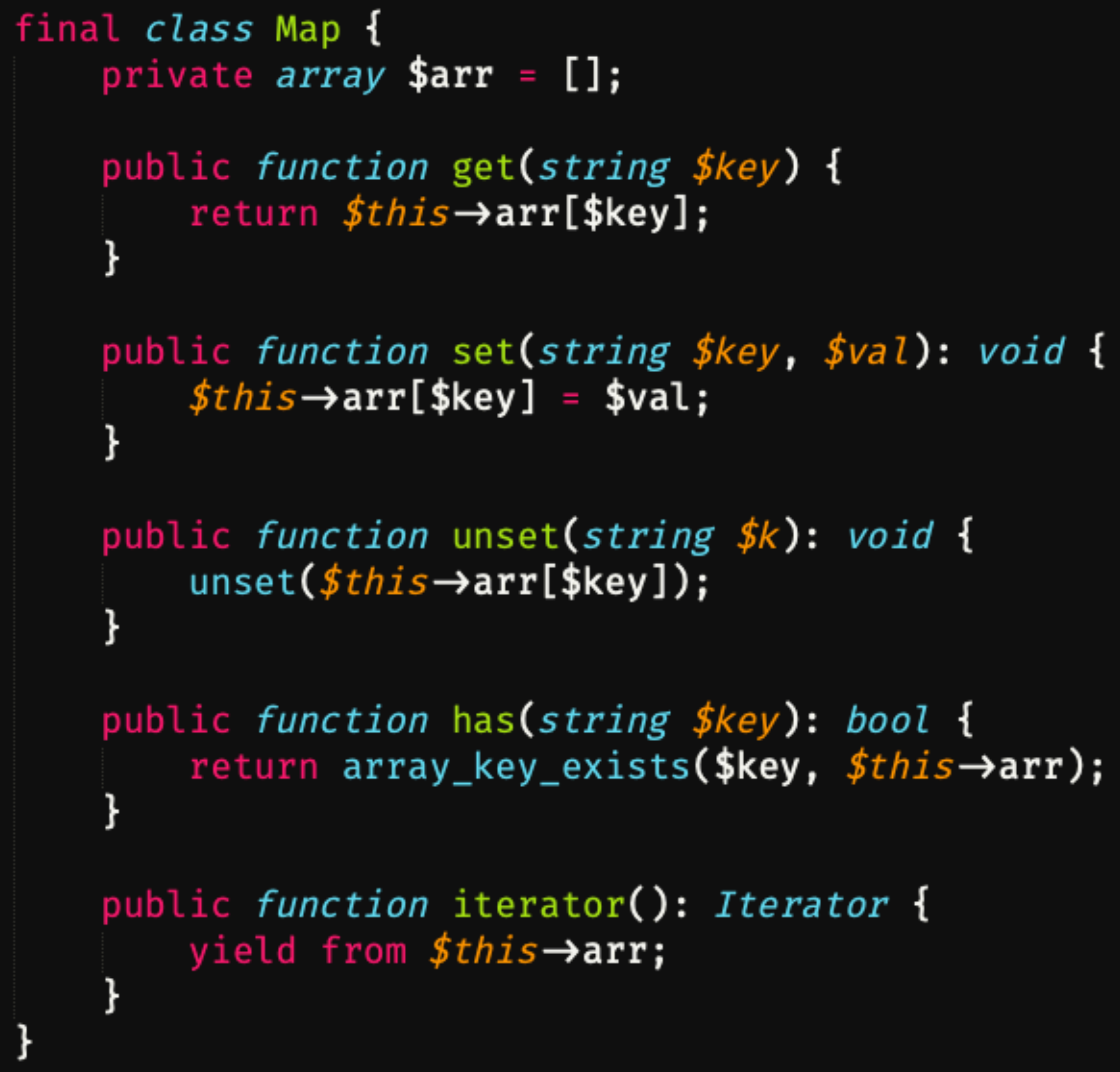
How can a single data structure be so flexible?



```php
final class Seq {
    private array $arr = [];

    public function get(int $idx): string {
        return $this->arr[$idx];
    }

    public function set(int $idx, $val): void {
        $this->arr[$idx] = $val;
    }

    public function push($val): void {
        $this->arr[] = $val;
    }

    public function insert(int $idx, $val): void {
        array_splice($this->arr, $idx, 0, $val);
    }

    public function remove(int $idx) {
        return array_splice($this->arr, $idx, 1)[0];
    }

    public function iterator(): Iterator {
        yield from $this->arr;
    }
}
```



```php
final class Set {
    private array $arr = [];

    public function add(string $val): void {
        $this->arr[$val] = true;
    }

    public function remove(string $val): void {
        unset($this->arr[$val]);
    }

    public function has(string $val): bool {
        return array_key_exists($this->arr, $val);
    }

    public function iterator(): Iterator {
        foreach ($this->arr as $key => $val) {
            yield $key;
        }
    }
}
```



```php
final class Map {
    private array $arr = [];

    public function get(string $key) {
        return $this->arr[$key];
    }

    public function set(string $key, $val): void {
        $this->arr[$key] = $val;
    }

    public function unset(string $k): void {
        unset($this->arr[$key]);
    }

    public function has(string $key): bool {
        return array_key_exists($key, $this->arr);
    }

    public function iterator(): Iterator {
        yield from $this->arr;
    }
}
```

Arrays: *Flexibility*

float and ***bool*** keys are converted to *integer.*

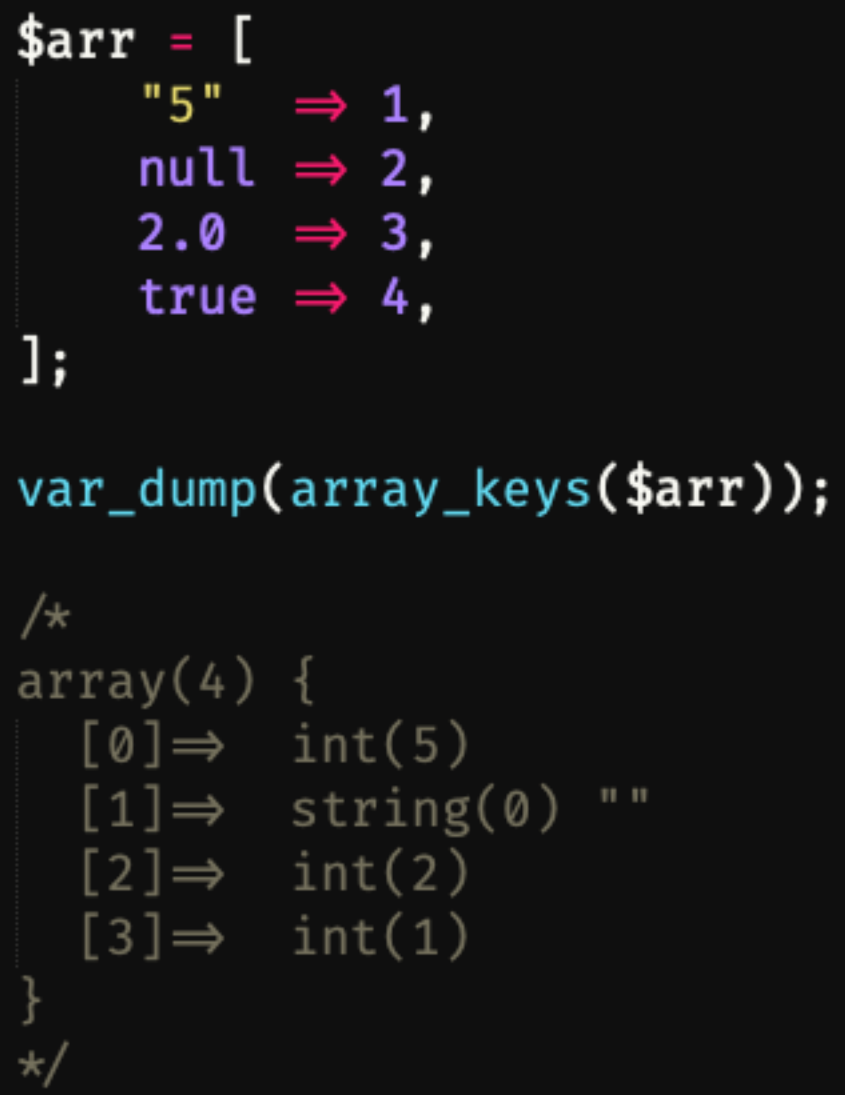
null is converted to a blank *string* key.

object can not be used as a key.

Sets are therefore restricted to contain **only *string*** and ***integer***.

Numeric strings are converted to *integers* when used as keys.

We have to explicitly **cast** to ***string*** to avoid this edge-case.



```php
$arr = [
    "5"  => 1,
    null => 2,
    2.0  => 3,
    true => 4,
];

var_dump(array_keys($arr));

/*
array(4) {
  [0]=>  int(5)
  [1]=>  string(0) ""
  [2]=>  int(2)
  [3]=>  int(1)
}
*/
```

Arrays: *Flexibility*

Flexibility increases complexity.

Flexibility requires more work (information, invariants).

Work is wasted when flexibility is not utilized.

For example, many situations do not require order maintenance.

Caches, lookup tables, ...

Single responsibility principle, modularity, composition.

Optimizing for everything optimizes for nothing!

Recap: *zval*

A *zval* is an internal container for all values in PHP. (C struct)

The name extends from "zend value".

Encapsulates a raw C union type (basically **raw bytes**), as well as PHP **type information** *(active type, how to interpret the bytes).*

There is also an additional **extra** field. (an unsigned integer)

The size of a *zval* is **16 bytes**.

Recap: *stdClass*

stdClass is the basic general **object** class.

Internally, class properties are stored using an array.

PHP could have used { } in the same way Javascript does.

Syntactic sugar, the underlying data structure is the same.

Some subtle differences, not a practical alternative.

Arrays: *Structure*

The internal structure of arrays consists of 2 major components:

An allocation of **buckets**, and an allocation of **hash indexes.**

A *bucket* contains a *zval*, an unsigned integer *hash*, and a pointer to a string (which is used when the key-value pair is associative).

The *hash index* guides the lookup to a bucket.

Arrays use the *extra* field in the *zval* for collision resolution.

Arrays maintains allocation **size**, the number of buckets **used**, and the **next** free slot in the bucket allocation.

Arrays: *Structure*

What happens when the bucket allocation is full?

What happens when two keys produce the same hash?

Some basic operations:

- **set** a key to a value
- **get** a value using a key
- **unset** a key
- **push** a value (append)
- **foreach**

How much memory do we need per key-value pair?

(16 + 8 + 8) + (4) = 36 bytes per column. **36 ~ 72 bytes**

Arrays: *Complexity*

random access	O(1), but can be O(n) in some cases.
array_push	O(1), amortized!
array_pop	O(1)
array_unshift	O(n)
array_shift	O(n)
array_merge	O(n)
array_keys	O(n)
array_reverse	O(n)
array_unique	$O(n^2)$
in_array	O(n)

Arrays: *Persistence*

Arrays use **copy-on-write**.

When an array is referenced more than once (shared), an update will first **copy** the array, then apply the update to the copy.

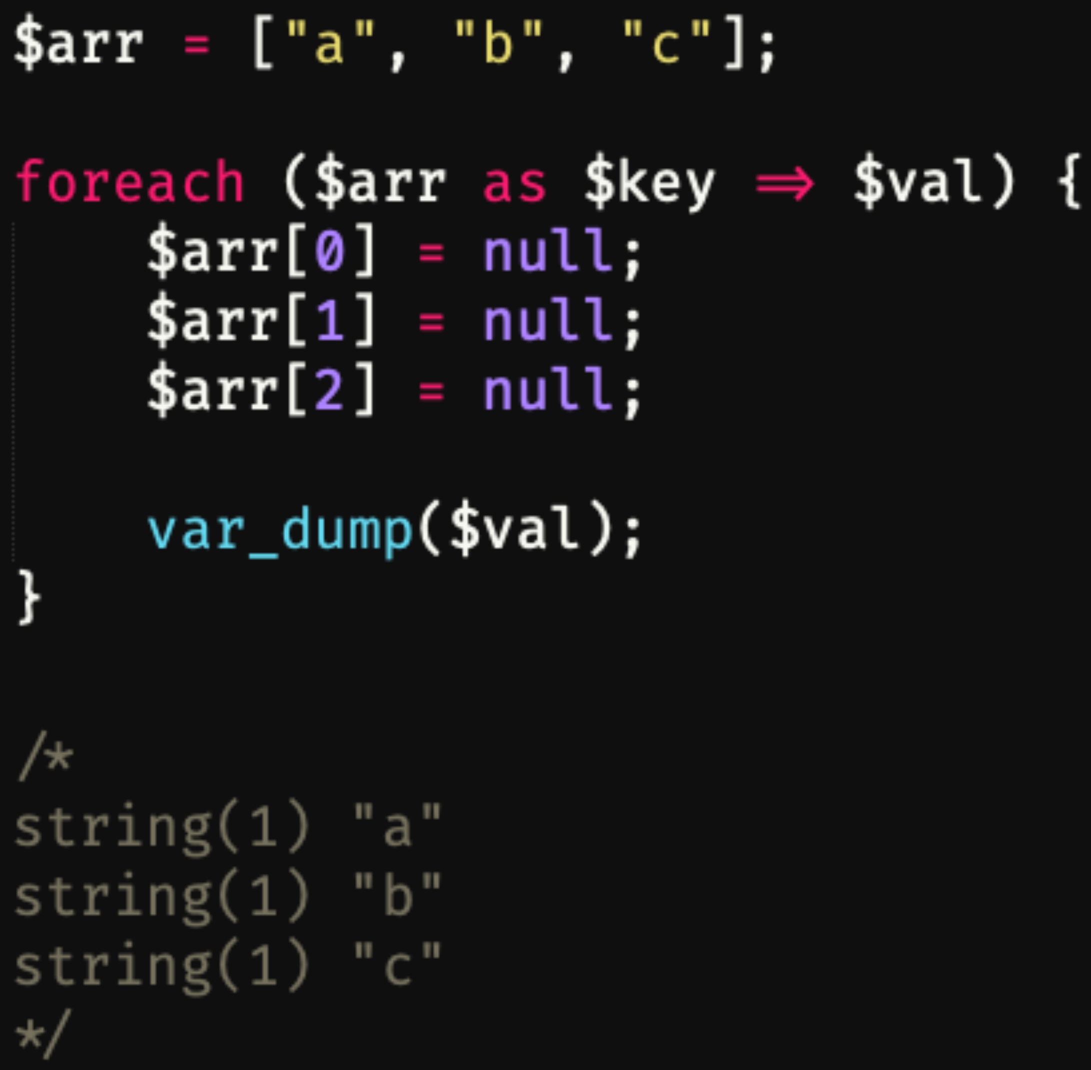
Copying an array is **O(n) --** we have to copy the entire allocation.

If the array is only **read**, no copying will be done.

Feels like **pass-by-value**.

```php
$a = ["x"];
debug_zval_dump($a);
/*
array(3) refcount(2){
  [0]=> string(1) "x" refcount(1)
}
*/

$b = $a; // Shallow copy!
debug_zval_dump($a);
/*
array(3) refcount(3){
  [0]=> string(1) "x" refcount(1)
}
*/

$b[] = "y"; // 1. Replace $b with a copy of $a
            // 2. Push "y" into $b
            // 3. $b is now a new version of $a

debug_zval_dump($a);
/*
array(3) refcount(2){
  [0]=> string(1) "x" refcount(1)
}
*/
```

```php
$a = ["a", "b", "c"];
$b = $a;

foreach ($b as $key => $val) {
    var_dump($val);
}

/*
string(1) "a"
string(1) "b"
string(1) "c"
*/
```

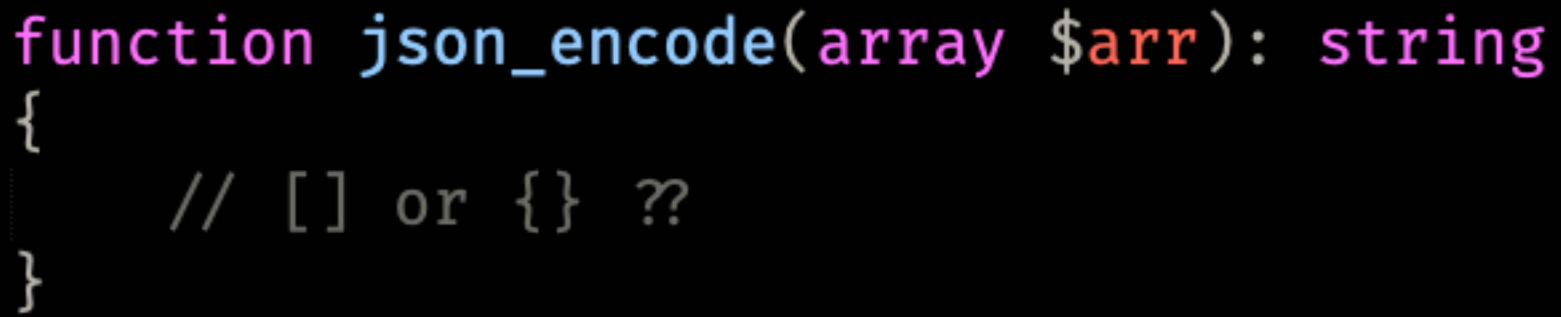


```php
$arr = ["a", "b", "c"];

foreach ($arr as $key => $val) {
    $arr[0] = null;
    $arr[1] = null;
    $arr[2] = null;

    var_dump($val);
}

/*
string(1) "a"
string(1) "b"
string(1) "c"
*/
```

Arrays: *Semantics*

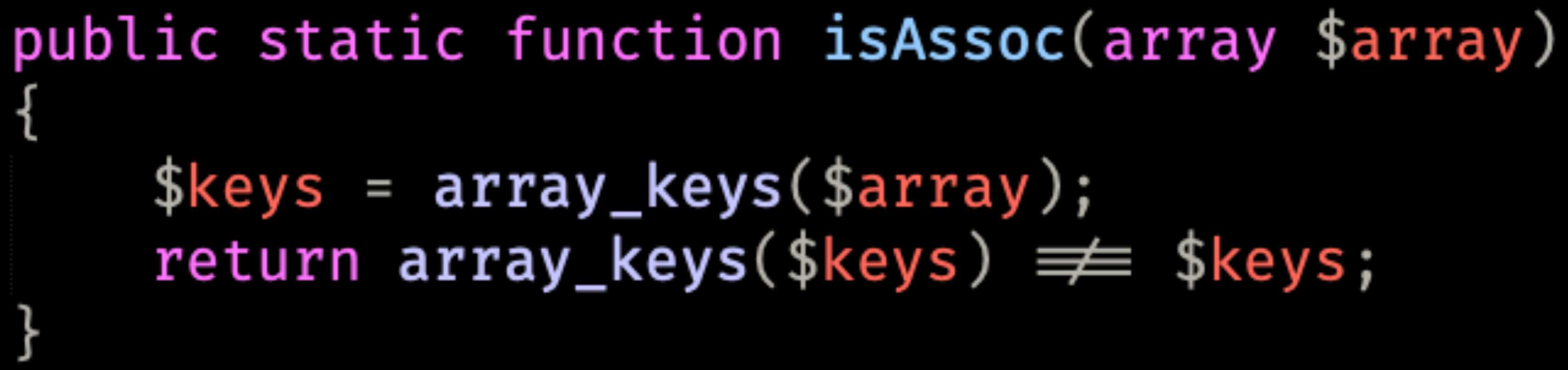
How do we know if an array is **linear** or **associative**?

We have to **inspect** the array or infer from **context**.



```
function json_encode(array $arr): string
{
    // [] or {} ??
}
```

What about Laravel's **Arr::isAssoc** ?

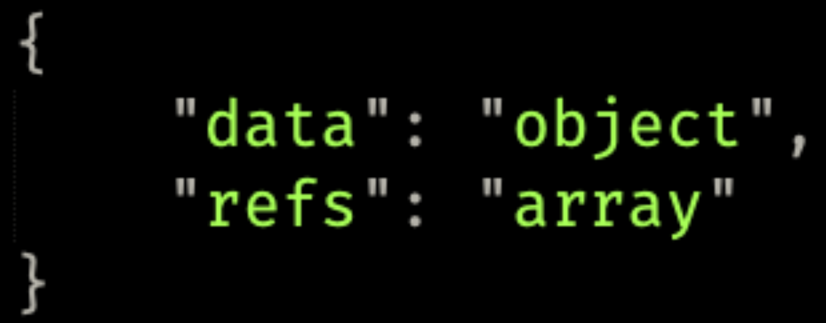


```
public static function isAssoc(array $array)
{
    $keys = array_keys($array);
    return array_keys($keys) !== $keys;
}
```

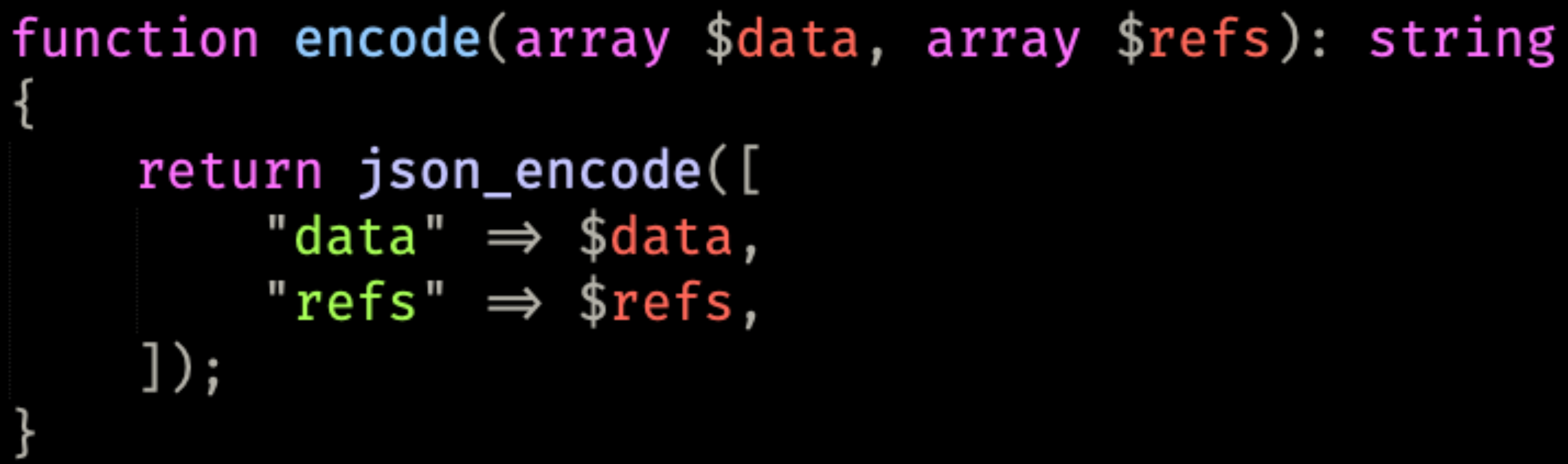
It's O(n) !!

Arrays: *Semantics*

Consider the following JSON schema:



```
{
    "data": "object",
    "refs": "array"
}
```



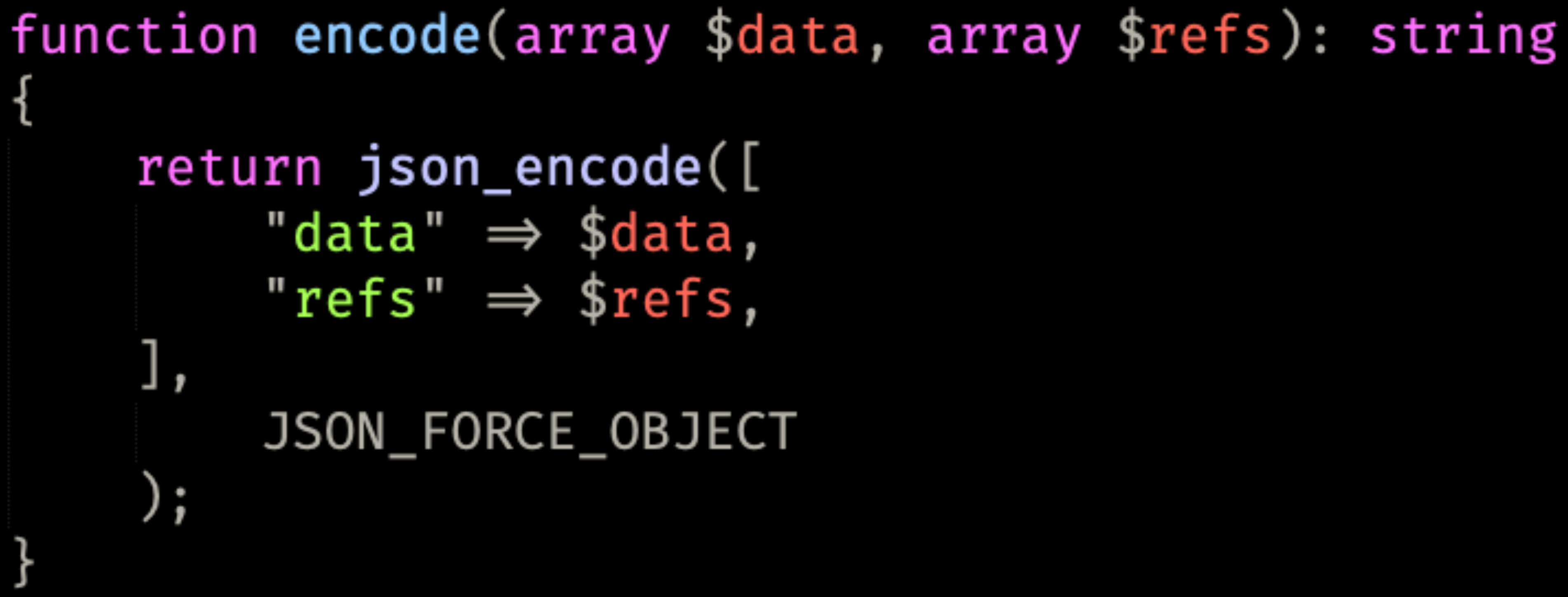
```
function encode(array $data, array $refs): string
{
    return json_encode([
        "data" => $data,
        "refs" => $refs,
    ]);
}
```

What happens when `$data` is empty?

An empty array is assumed to be linear. `json_encode([]); // "[]"`

Arrays: *Semantics*

We can use `JSON_FORCE_OBJECT` to convert the "[ ]" into a "{ }"



```
function encode(array $data, array $refs): string
{
    return json_encode([
        "data" => $data,
        "refs" => $refs,
    ],
        JSON_FORCE_OBJECT
    );
}
```

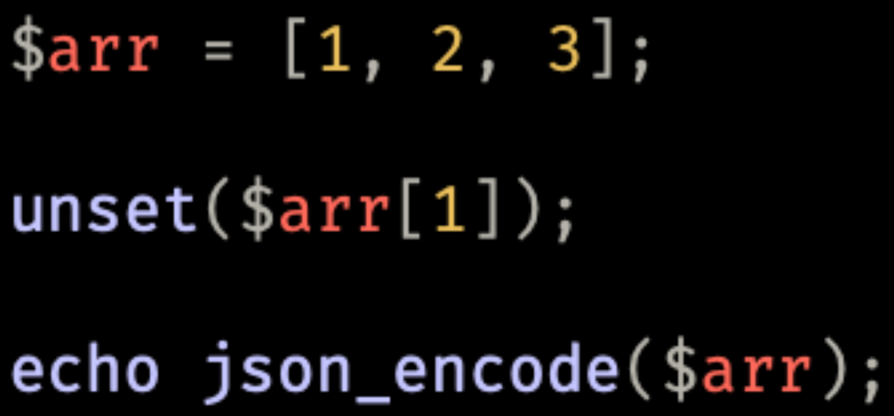
But this converts **all** arrays to objects!

We have to explicitly **cast** `$data` to (object).

This is not the end of the world, but does add responsibility.

Arrays: *Semantics*

What happens when you **unset** an index of a linear array?



```
$arr = [1, 2, 3];

unset($arr[1]);

echo json_encode($arr);
```

It becomes associative! `// {"0":1,"2":3}`

Can we put it back? `$arr[1] = 2;`

It's a mess. `// {"0":1,"2":3,"1":2}`

Can we do better?

php-ds: *Introduction*

ds is a PHP language **extension** that provides low-level C implementations of some fundamental data structures.

First release was in 2016. ~10,000 monthly downloads.

Motivations

- Provide **semantic** value without sacrificing performance.
- Provide **specialized** containers that outperform arrays.
- Provide **standard** interfaces for collections.

github.com/php-ds

php-ds: *Features*

Vector

Linear, sequential, low memory.
O(1) random access, *push, pop.*

Deque

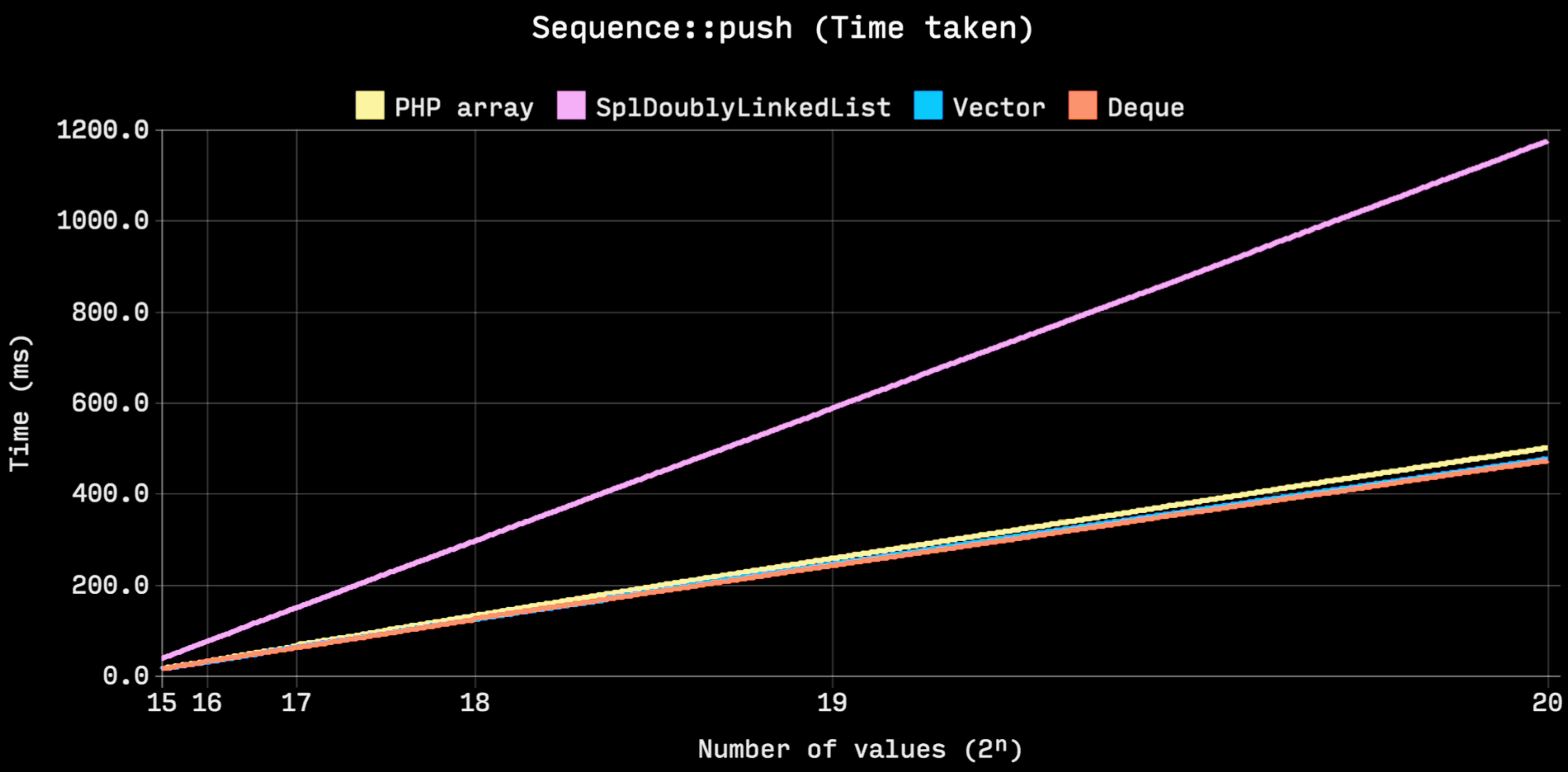
Linear, sequential.
O(1) random access, *push, pop, shift, unshift.*
"Double ended queue".

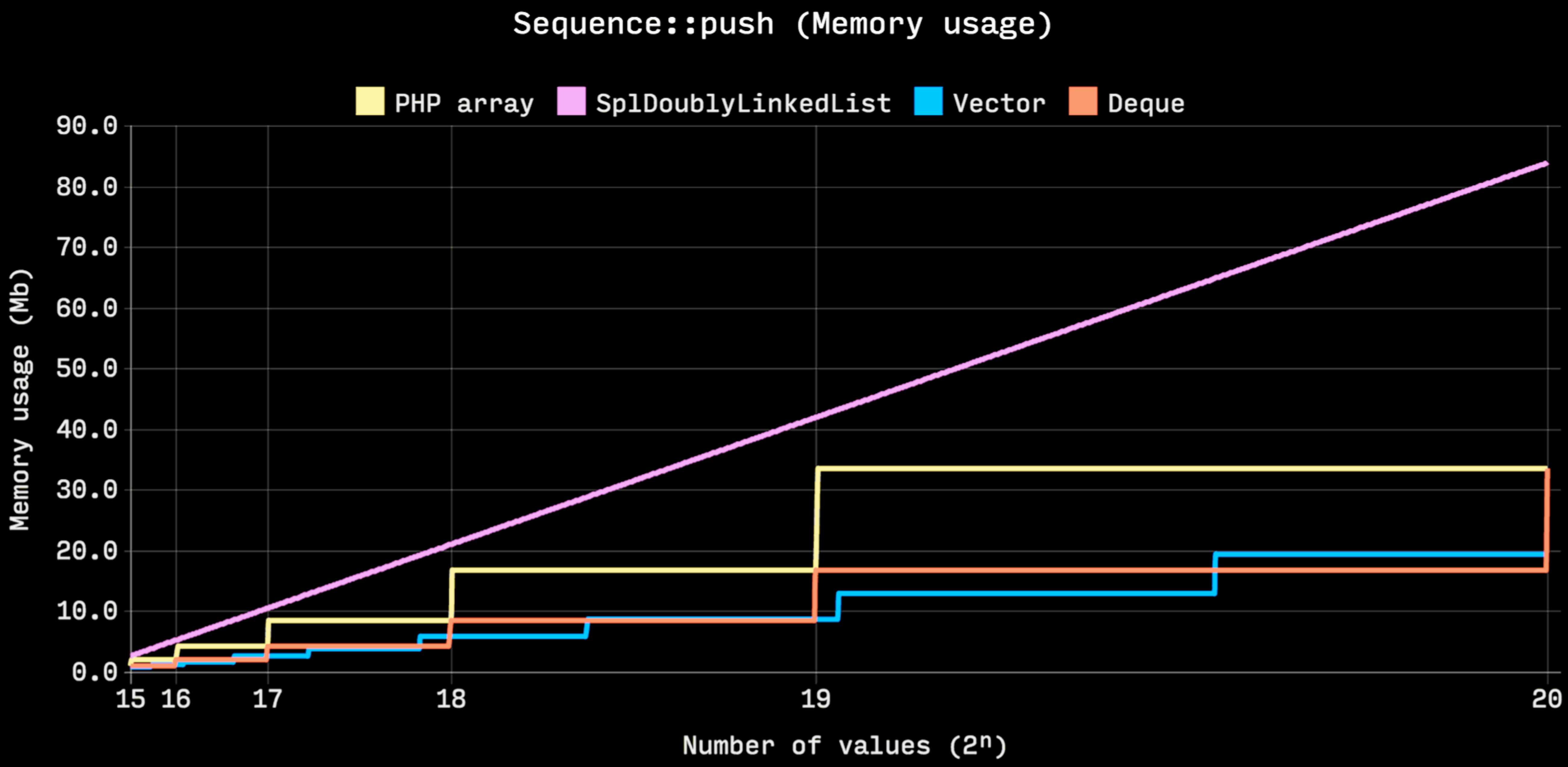
Set

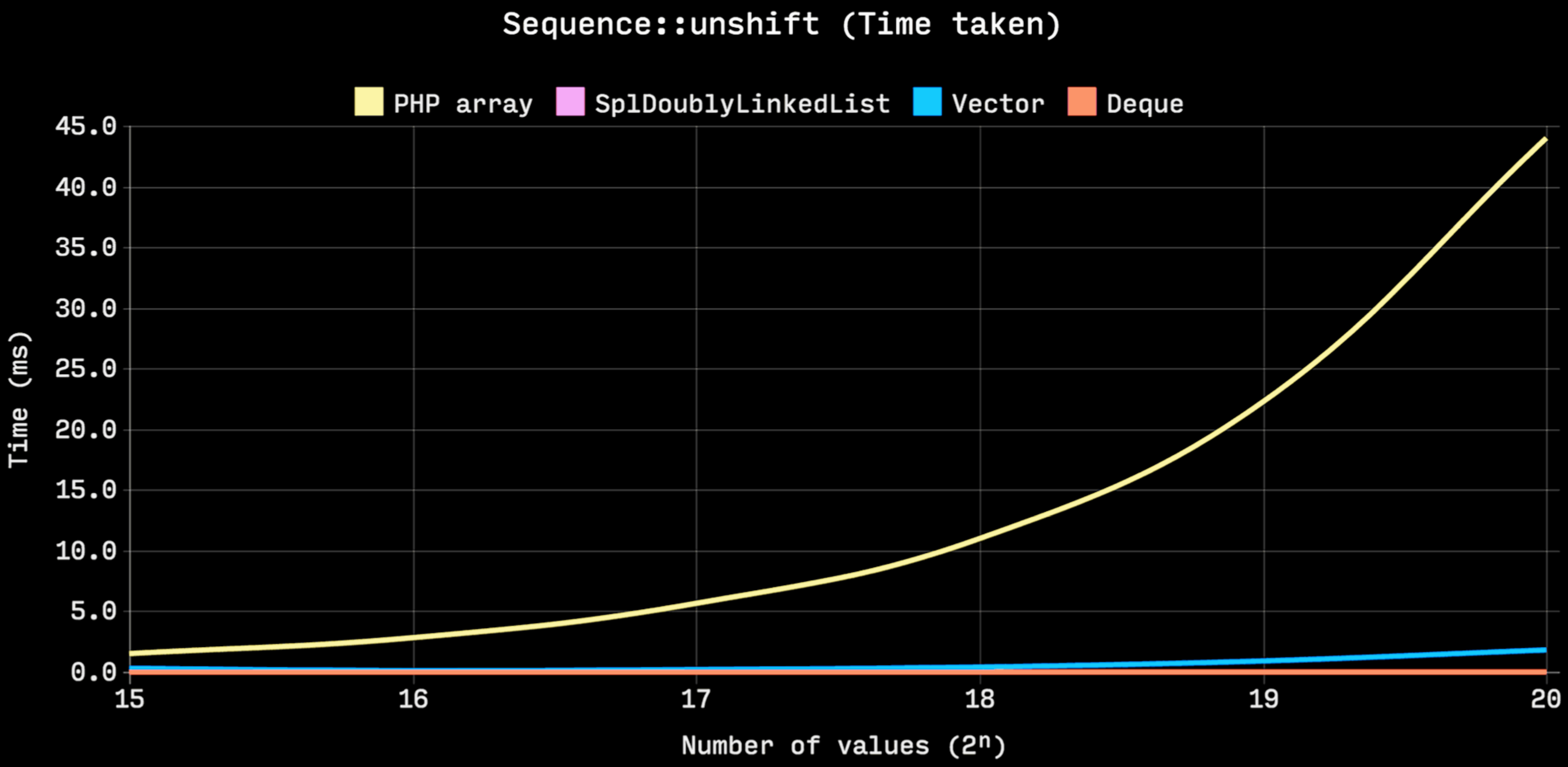
Linear, sequential, **equivalent in performance to arrays**.
Supports values of any type.
O(1) *add, remove, has.*

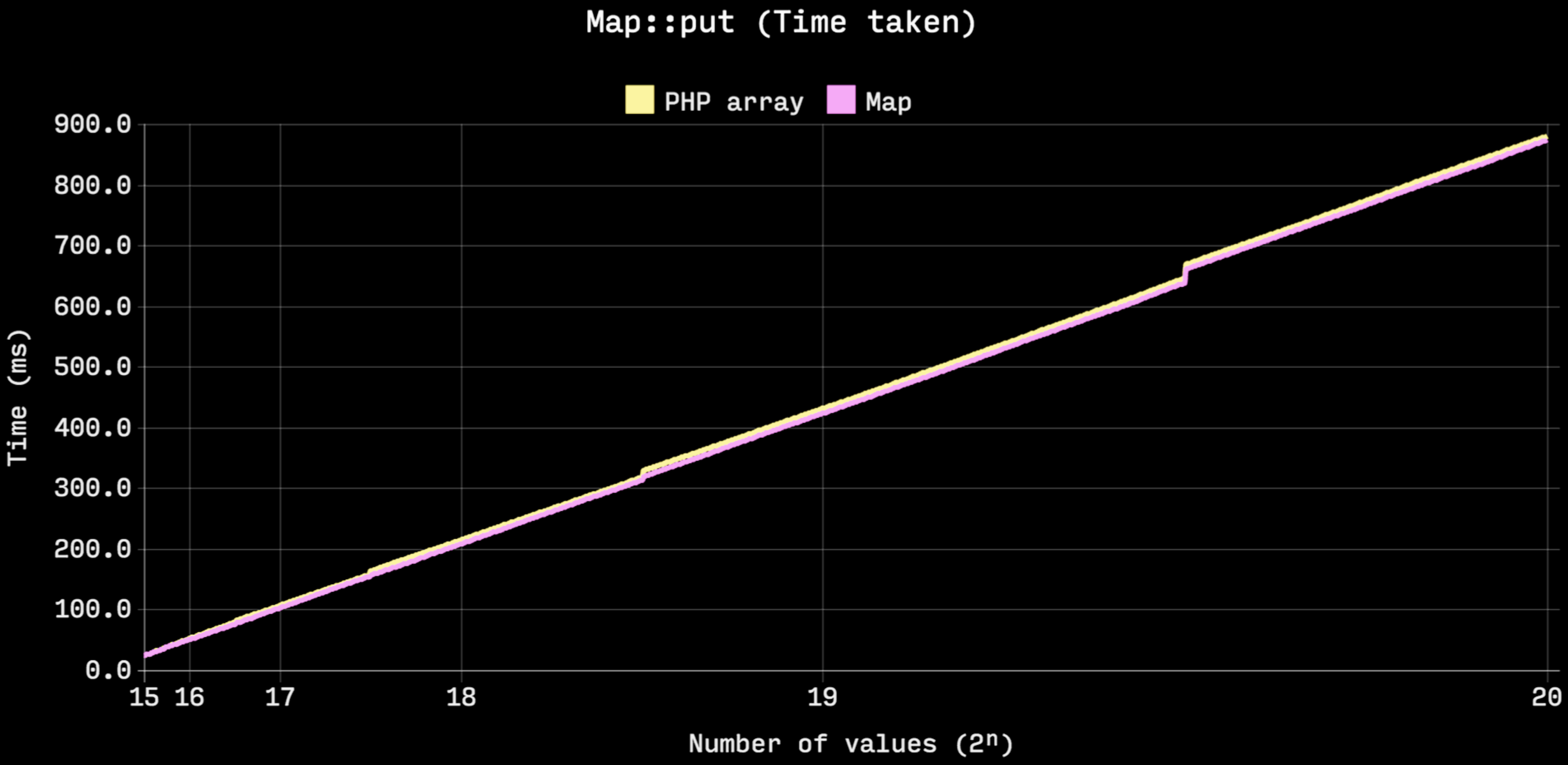
Map

Associative, **equivalent in performance to arrays**.
Supports keys of any type.
O(1) *put, remove, has.*

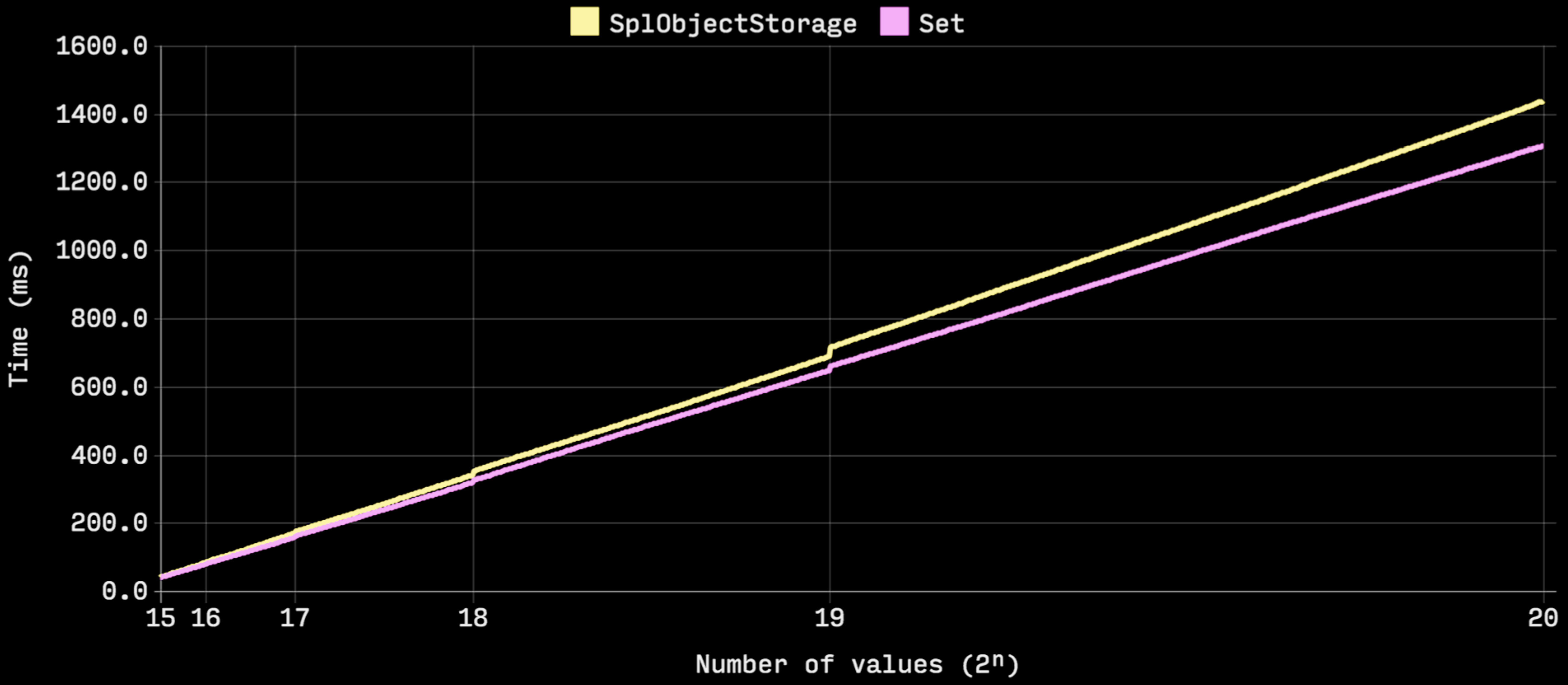

Sequence::push (Time taken)
PHP array
SplDoublyLinkedList
Vector
Deque
Time (ms)
1200.0
1000.0
800.0
600.0
400.0
200.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)

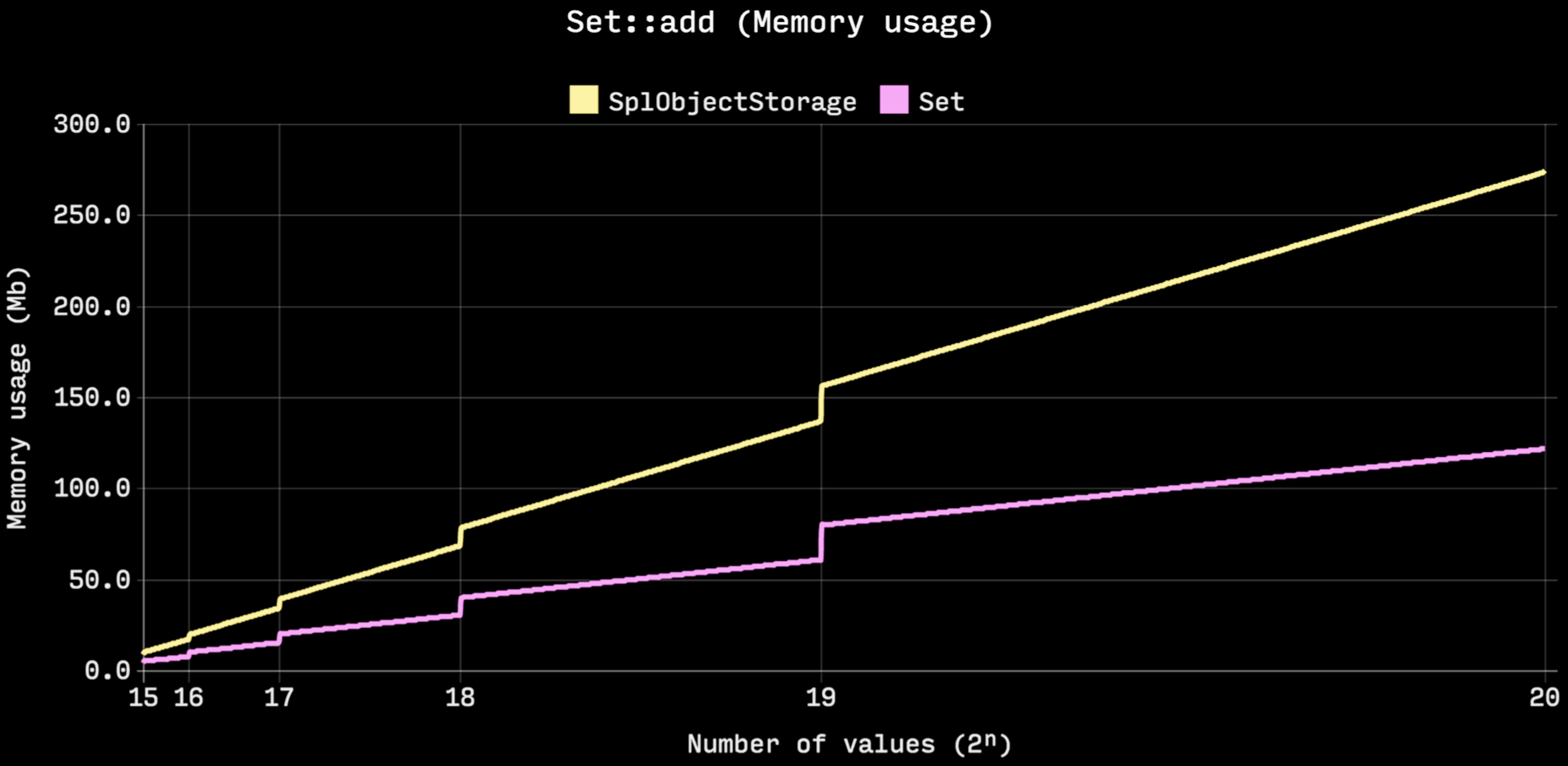

Sequence::push (Memory usage)
PHP array
SplDoublyLinkedList
Vector
Deque
Memory usage (Mb)
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)

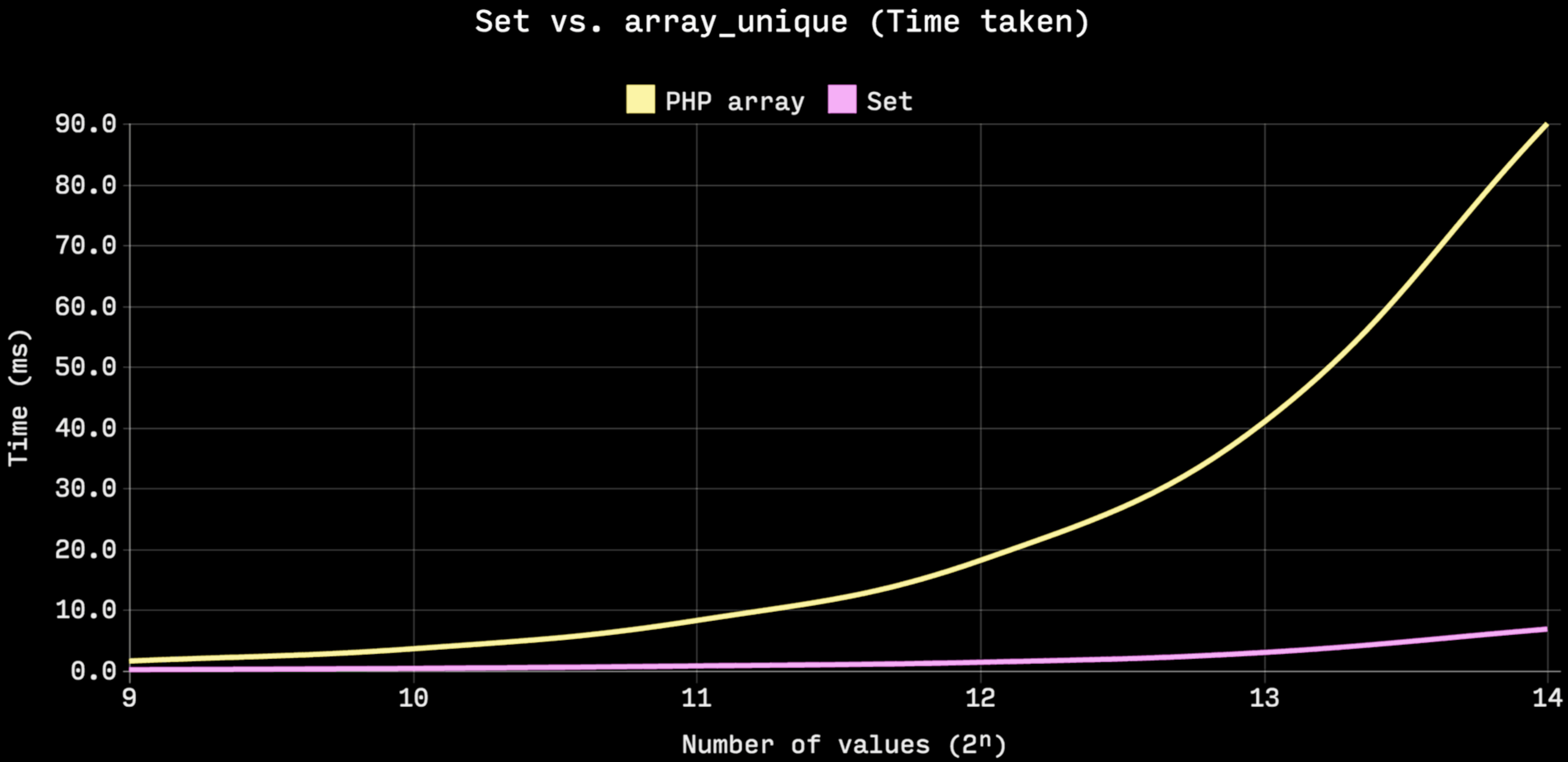

Sequence::unshift (Time taken)
PHP array
SplDoublyLinkedList
Vector
Deque
Time (ms)
45.0
40.0
35.0
30.0
25.0
20.0
15.0
10.0
5.0
0.0
15
16
17
18
19
20
Number of values (2ⁿ)


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

Set::add (Time taken)




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 (2^n)


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

php-ds: *What's next?*

We can **avoid O(n) copying** by partially sharing memory between instances that have data in common.

Many cases require copying only **O(1)** values per update.

Most other cases only **O(log n).**

Reduces garbage collection volume and memory allocation.

Allows for fast, efficient **immutable** data structures in PHP.

Indirectly makes **functional programming** viable in PHP.