Why and how to implement efficient data structures to use with node.js or in the browser?
Who am I?

Guillaume Plique

alias Yomguithereal on both Github and Twitter.

Research engineer working for Sciences Po's médialab.
What's a data structure?
«Web development is not real development and is henceforth easier.»

Someone wrong on the Internet.
«Web development is trivial and web developers don't need fancy data structures or any solid knowledge in algorithmics.»

Someone also wrong (and pedant) on the Internet.
Don't we already have fully satisfying data structures in JavaScript?

- **Array** ➞ lists of things
- **Object** ➞ key-value associations
- **Map** and **Set** with ES6
Why would we want other data structures in JavaScript?
Convenience and bookkeeping
• A MultiSet

// How about changing this:
const counts = {};

for (const item in something) {
    if (!(item in counts))
        counts[item] = 0;
    counts[item]++;
}

// Into this:
const counts = new MultiSet();

for (const item in something)
    counts.add(item);
• **Complex structures: a Graph**

Sure, you can "implement" graphs using only `Array` and `Object™`. But:

• Lots of bookkeeping (multi-way indexation)
• Wouldn't it be nice to have a legible interface?
Examples taken from the **graphology** library:

```javascript
const graph = new Graph();

// Finding specific neighbors
const neighbors = graph.outNeighbors(node);

// Iterating over a node's edges
graph.forEachEdge(node, (edge, attributes) => {
    console.log(attributes.weight);
});
```
- Sometimes Arrays and Objects are not enough
• More than just tacky website candy
  • We process data on the client nowadays.
  • Node.js became a thing.
  • Some algorithms cannot be efficiently implemented without custom data structures (Dijkstra or Inverted Index for full text search etc.).
• The QuadTree
The QuadTree
What are the challenges?
• Interpreted languages are far from the metal
• No control over memory layout
• No control over garbage collection
• JIT & optimizing engines such as Gecko / V8
Benchmarking code accurately is not easy.
It does not mean we cannot be clever about it.
• Implementation tips
• Time & memory performance
• **Minimizing lookups**

"Hashmap" lookups are costly.

```javascript
// You made 2 lookups
Graph.prototype.getNodeAttribute = function(node, data) {
    if (this._nodes.has(node))
        throw Error(...);

    const data = this._nodes.get(node);

    return data[name];
};
```
// You made only one
Graph.prototype.getNodeAttribute = function(node, data) {
  const data = this._nodes.get(node);

  if (typeof data === 'undefined')
    throw Error(...);

  return data[name];
};
The engine is clever. But not *that* clever. (It improves frequently, though...)

The «*let's code badly, the engine will clean up my mess*» approach will not work.
• Creating objects is costly

• Avoid allocating objects.
• Avoid /(?:re-)?creating/ regexes.
• Avoid nesting functions whenever possible.
// BAD!
const test = x => /regex/.test(x);

// GOOD!
const REGEX = /regex/;
const test = x => REGEX.test(x);

// BAAAAAD!
function(array) {
  array.forEach(subarray => {

    // You just created one function per subarray!
    subarray.forEach(x => console.log(x));

  });
}
Mixing types is bad

// Why do you do that?
// If you are this kind of person, can we meet?
// I really want to understand.
const array = [1, 'two', '3', /four/, {five: new Date()}];
• The poor man's malloc

• Byte arrays are fan-ta-stic.
• Byte arrays are light.
• You can simulate typed memory allocation: `Uint8Array`, `Float32Array` etc.
• Implement your own pointer system!

And have your very own "C in JavaScript"™.
A linked list (with pointers):
--------------------------
head -> (a) -> (b) -> (c) -> Ø

// Using object references as pointers
function ListNode(value) {
    this.next = null;
    this.value = value;
}

// Changing a pointer
node.next = otherNode;
A linked list (rolling our own pointers):

head = 0
values = [a, b, c]
next = [1, 2, 0]

// Using byte arrays (capacity is fixed)
function LinkedList(capacity) {
  this.head = 0;
  this.next = new Uint16Array(capacity);
  this.values = new Array(capacity);
}

// Changing a pointer;
this.next[nodeIndex] = otherNodeIndex;
• **Let's build a most efficient LRU Cache!**

  • An object with maximum number of keys to save up some RAM.
  • If we add a new key and we are full, we drop the Least Recently Used one.
  • Useful to implement caches & memoization.
A ~doubly~ linked list:

- head = 0
- tail = 2
- next = [1, 2, 0]
- prev = [0, 1, 2]

Same as (with pointers):

head -> (a) <-> (b) <-> (c) <-> tail

A map to pointers & values:

items = {a: 0, b: 1, c: 2}
values = [a, b, c]
<table>
<thead>
<tr>
<th>name</th>
<th>set</th>
<th>get1</th>
<th>update</th>
<th>get2</th>
<th>evict</th>
</tr>
</thead>
<tbody>
<tr>
<td>mnemonist-object</td>
<td>15314</td>
<td>69444</td>
<td>35026</td>
<td>68966</td>
<td>7949</td>
</tr>
<tr>
<td>tiny-lru</td>
<td>6530</td>
<td>46296</td>
<td>37244</td>
<td>42017</td>
<td>5961</td>
</tr>
<tr>
<td>lru-fast</td>
<td>5979</td>
<td>36832</td>
<td>32626</td>
<td>40900</td>
<td>5929</td>
</tr>
<tr>
<td>mnemonist-map</td>
<td>6272</td>
<td>15785</td>
<td>10923</td>
<td>16077</td>
<td>3738</td>
</tr>
<tr>
<td>lru</td>
<td>3927</td>
<td>5454</td>
<td>5001</td>
<td>5366</td>
<td>2827</td>
</tr>
<tr>
<td>simple-lru-cache</td>
<td>3393</td>
<td>3855</td>
<td>3701</td>
<td>3899</td>
<td>2496</td>
</tr>
<tr>
<td>hyperlru-object</td>
<td>3515</td>
<td>3953</td>
<td>4044</td>
<td>4102</td>
<td>2495</td>
</tr>
<tr>
<td>js-lru</td>
<td>3813</td>
<td>10010</td>
<td>9246</td>
<td>10309</td>
<td>1843</td>
</tr>
</tbody>
</table>

Bench [here](#) - I masked libraries which are not LRU per se.
● Function calls are costly

Everything is costly. Life is harsh.

This means that rolling your own stack will always beat recursion.
// Recursive version - "easy"
function recurse(node, key) {
    if (key < node.value) {
        if (node.left)
            return recurse(node.left, key);

        return false;
    }

    else if (key > node.value) {
        if (node.right)
            return recurse(node.right, key);

        return false;
    }

    return true;
}
// Iterative version - more alien but faster, mileage may vary

function iterative(root, key) {
    const stack = [root];
    while (stack.length) {
        const node = stack.pop();
        if (key < node.value) {
            if (node.left)
                stack.push(node.left);
            else
                break;
        } else if (key > node.value) {
            if (node.right)
                stack.push(node.right);
            else
                break;
        }
        return true;
    }
    return false;
}
• **What about wasm etc.?**

Lots of shiny options:

1. asm.js
2. WebAssembly
3. Native code binding in Node.js
Communication between those and JavaScript has a cost that negates the benefit.

This is only viable if you have long running code or don't need the bridge between the layer and JavaScript.
Parting words
• Yes, optimizing JavaScript is hard.
But it does not mean we cannot do it.
• Most tips are applicable to every high-level languages.
But JavaScript has its very own kinks.

The **ByteArray** tips absolutely don't work in Python.

It's even slower if you use **numpy** arrays. (you need to go full native).
The gist

To be efficient your code must be \textit{statically interpretable}.

If you do that:

1. The engine will have \textbf{no hard decisions} to make
2. And will safely choose the most aggressive optimization paths
Rephrased

Optimizing JavaScript = squinting a little and pretending really hard that:

1. The language has static typing.
2. That the language is low-level.
• **Associative arrays are the next frontier**

For now, there is no way to beat JavaScript's objects and maps when doing key-value association.

*Yet...*
• So implement away!
• **References**

Examples were taken from the following libraries:

- [mnemonist](yomguithereal.github.io/mnemonist)
- [graphology](graphology.github.io)
- [sigma.js](sigmajs.org)
Thanks!