Implementing Heaps in JavaScript

Learn what Heaps are and what sort of problems they help solve

Ankita Masand Follow
Oct 2, 2019 · 10 min read ★

Most of the programming languages support some specific data types, for example, `int`, `string`, `boolean`, etc. We can define our custom data type for storing groups of data and this data type can also have functions/operations. These functions/operations can be applied to the data points to get meaningful results. The logical model for the custom data type is called *Abstract Data Type (ADT)* and the physical implementation of these types is called a D*ata Structure*. Data Structures are the most fundamental unit in computer science and it is very important to use the correct data structure for solving problems efficiently. Some of the popular data structures are *Arrays*, *Stacks*, *Queue*, *LinkedList*, *Trees*, *Heaps*, etc. Unlike other high-level languages, most of these data types don't come bundled with the native JavaScript runtime.

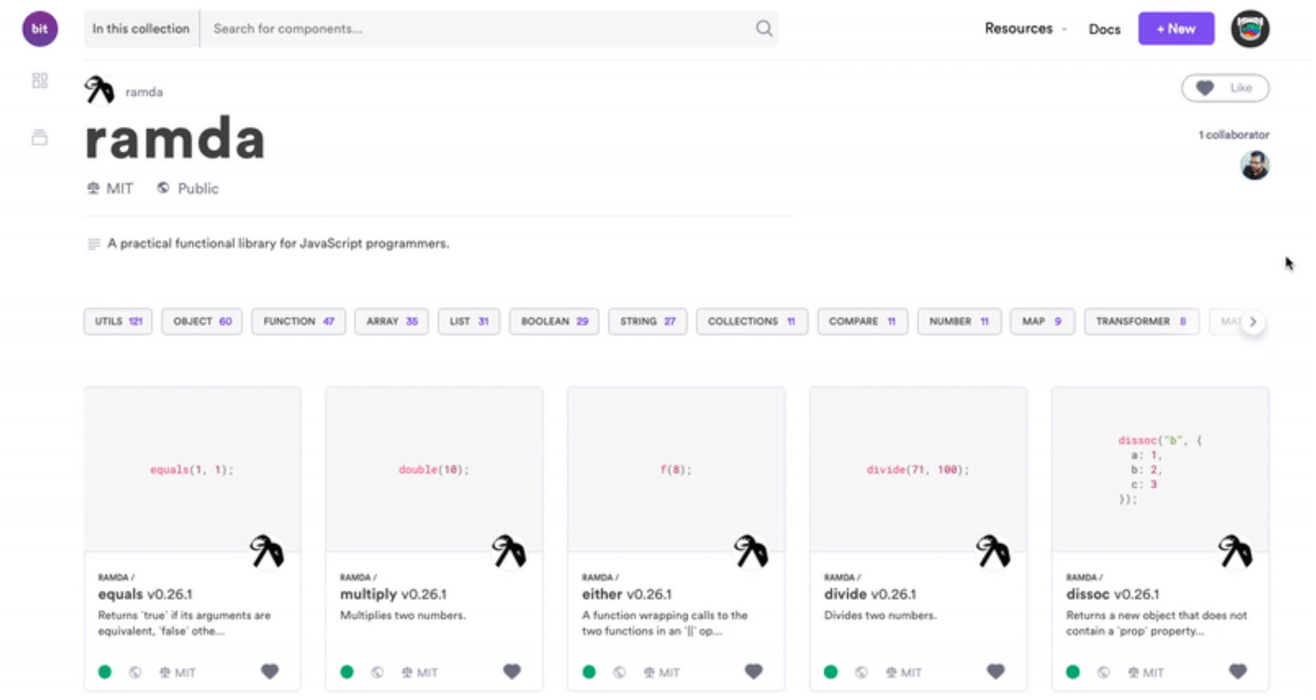
In this article, we're going to look at one of the interesting data structures — Heaps!

> Heaps data structure, unlike Object, Map, and Set is not supported natively in JavaScript.

We're going to implement heaps from scratch. But first, let's try to understand what heaps are and what sort of computer science problems can Heaps solve?

Tip: Share and reuse utility and UI components

Use **Bit** to easily publish, install and update small individual modules across different JavaScript projects. Don't install entire libraries and don't copy-paste code, when you can build faster with small reusable modules. Take a look.



ramda

A practical functional library for JavaScript programmers.

```
equals(1, 1);
```

RAMDA / **equals** v0.26.1
Returns `true` if its arguments are equivalent, `false` othe...

```
double(10);
```

RAMDA / **multiply** v0.26.1
Multiplies two numbers.

```
f(8);
```

RAMDA / **either** v0.26.1
A function wrapping calls to the two functions in an `||` op...

```
divide(71, 100);
```

RAMDA / **divide** v0.26.1
Divides two numbers.

```
dissoc("b", {
  a: 1,
  b: 2,
  c: 3
});
```

RAMDA / **dissoc** v0.26.1
Returns a new object that does not contain a `prop` property...

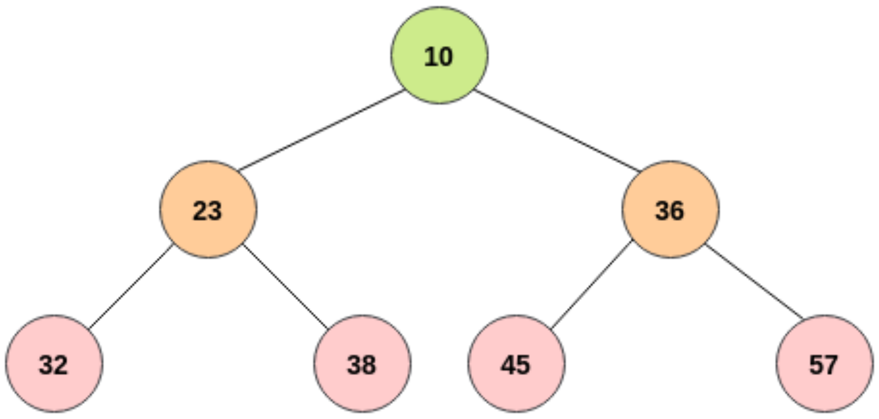
Bit: Easily reuse and sync small modules and utils across JS projects

What is a Heap data structure?

A heap is a tree-based data structure which is an almost complete tree that satisfies the heap property.

A complete tree is a tree in which every level, except possibly the last, is completely filled and all nodes are as far left as possible.

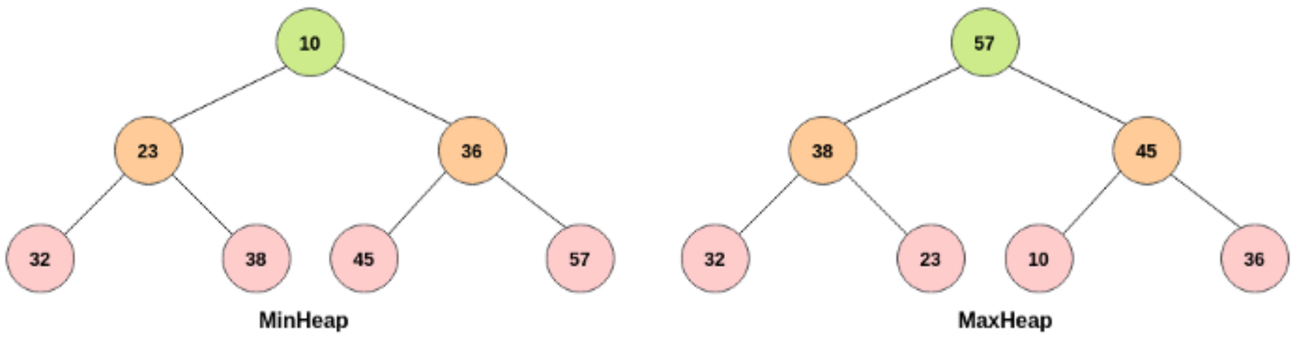
We'll get to the unknown heap property in a moment. Here's how a binary heap looks like:



Representation of a Binary Heap

A heap is essentially used to get the highest priority element at any point in time. There are two types of heaps, based on the heap property — *MinHeap* and *MaxHeap*.

MinHeap: The parent node is always less than the child nodes.
MaxHeap: The parent node is always greater than or equal to the child nodes.



Representation of MinHeap & MaxHeap

In *MinHeap*, the root node `10` is smaller than its two child nodes `23` and `36` while `23` and `36` are smaller than their respective child nodes.
In *MaxHeap*, the root node `57` is greater than its two child nodes `38` and `45` while `38` and `45` are greater than their respective child nodes.

Why do we need something like Heaps?

Heaps is primarily used for getting the minimum or the maximum value present in a heap in `O(1)` time. The linear data structures like *Arrays* or *LinkedList* can get you this value in `O(n)` time while non-linear data structures like Binary Search Trees(BST) can get you this value in `O(log n)` time where `n` is the number of elements.

Here's the time complexity of various operations performed on a heap with `n` elements:
Access the min/max value: `O(1)`
Inserting an element: `O(log n)`
Removing an element: `O(log n)`

Heaps make it blazing fast to access the priority-level elements. The Priority Queue data structure is implemented using Heaps. As the name suggests, you can access elements on a priority basis in `O(1)` time using a Priority Queue. It is commonly used in Dijkstra's

Algorithm, Huffman Coding. Don't worry if you don't know these algorithms yet! We'll cover them in detail in the next set of articles.

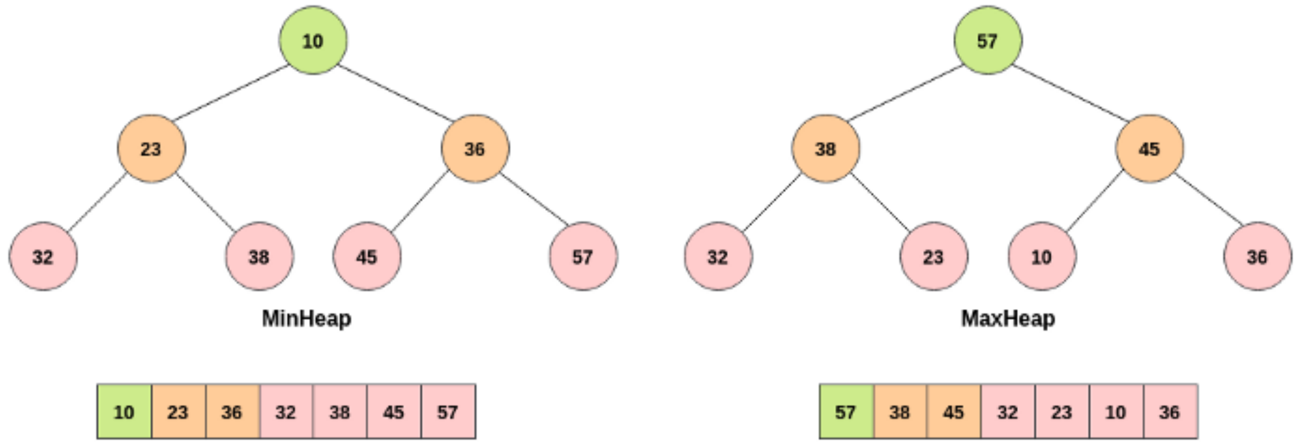
Heaps helps in quicker access to the min/max elements. Got it! But why do we need these elements in the first place?

Here are some of the real-world use-cases of Heaps:

1. The Operating System uses heaps for scheduling jobs on a priority basis.
2. The producer-consumer model can be implemented using heaps to let consumer access the higher priority elements first. It is used in the implementation of Blocking Priority Queue.
3. Other applications include Order Statistics to find the kth-smallest/kth-largest element in an Array, in HeapSort Algorithm and Graph Algorithms such as Djiktra's to find the shortest path and Prim's minimal spanning tree algorithm.

How to implement Heaps?

We represent heaps as sort of trees but they're not stored as trees in the memory. Let's try to convert the heap representation in an array and see how it turns out:



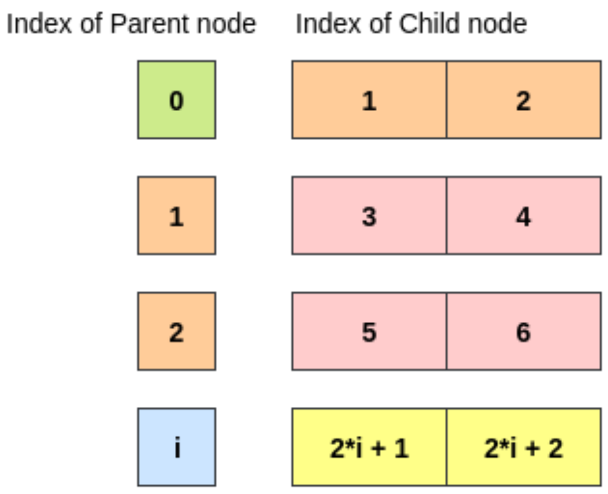
Implementing Binary Heaps using Arrays

Please note the order in which I've added the elements in an array. The element `10` is at position `0` and its two children are at position `1` and `2`. And, then I've added the child nodes of `23` – `32` & `38` and after these nodes, the child nodes of `36` are added. I've added the elements level by level. Same goes for the MaxHeap!

But why we follow this particular arrangement of filling the array level by level?

If you carefully observe, the minimum and the maximum elements are placed at the `0th` index in their respective arrays. We can access these elements in constant `O(1)` time.

If a parent is at `0th` index, its two child nodes are at `1st` and the `2nd` position in the array. Here's the relationship between the parent and the child nodes in a binary heap:



Relationship of array indices of the parent and the child node in a Binary Heap

From the above image, we can deduce that the child nodes of any element at index `i` are placed at indexes `2*i + 1` and `2*i + 2`. Also, we can go reverse and find out the parent node of any element at `i` using the formula `i/2`. Please note: this is applicable only for Binary Heaps.

Heaps can be implemented using Arrays or LinkedList but the common way of implementing them is by using Arrays. Now, let's get to the interesting bit and start implementing Heaps!

We're going to implement `MinHeap` and it will have methods for accessing the min element, inserting a new element and removing an element from the heap. Let's start by creating a `MinHeap` class:

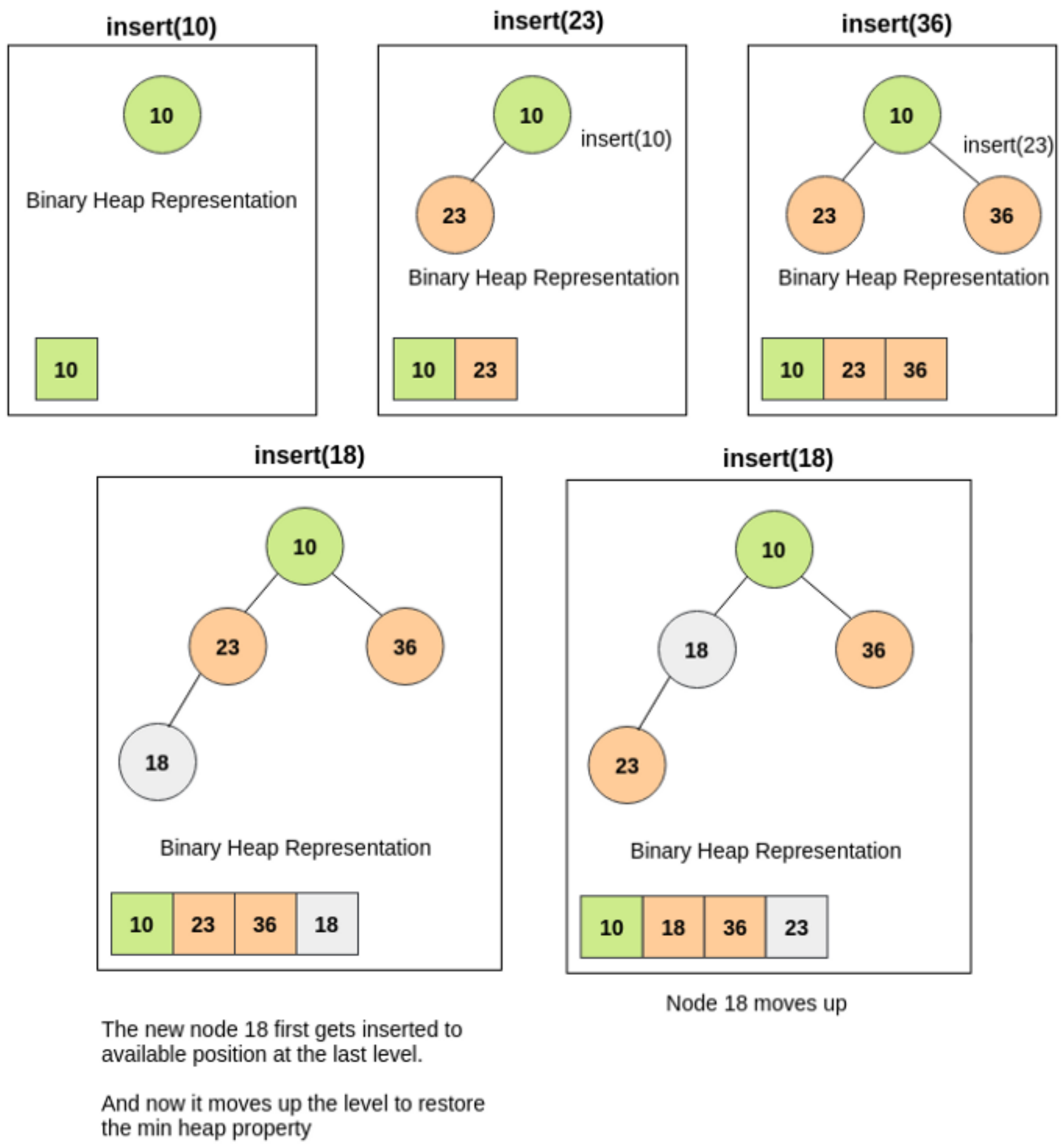
```javascript
class MinHeap {

    constructor () {
        /* Initialing the array heap and adding a dummy element at index 0 */
        this.heap = [null]
    }

    getMin () {
        /* Accessing the min element at index 1 in the heap array */
        return this.heap[1]
    }
}
```

min-heap-getMin.js hosted with ❤ by **GitHub**   **view raw**

There's nothing much going on here! I've created an array `heap` and initialized it with `null` at the `0th` index; the actual heap will start getting filled from the `1st` index. This is done just to make things easy to understand. `getMin` is a simple function that returns the very first element in the heap.

Time Complexity of accessing the min/max element in the heap is `O(1)`.

As mentioned earlier, heaps are complete trees except for the last level. The new nodes are inserted from left to right and the heap property is maintained on every insertion. Let's see this in action:


insert(10)
10
Binary Heap Representation
10
insert(23)
10
insert(10)
23
Binary Heap Representation
10 23
insert(36)
10
insert(23)
23
36
Binary Heap Representation
10 23 36
insert(18)
10
23
36
18
Binary Heap Representation
10 23 36 18
The new node 18 first gets inserted to available position at the last level.
And now it moves up the level to restore the min heap property
insert(18)
10
18
36
23
Binary Heap Representation
10 18 36 23
Node 18 moves up

Insertion in Binary Heap

The above image clearly explains the insertion in Binary Heaps but let me put it into words for better understanding.

`insert(10)` : Binary Heap is empty and we'd want to insert `10` as the first node. `10` gets added at the first position. Easy!

`insert(23)` : Heaps are filled from left to right. Now, `23` gets added as the left child of `10`. And since `10` (parent node) is already less than the child node, we don't have to do anything more here!

`insert(36)` : Node `36` gets added as the right child of `10`. We don't have to shuffle any nodes here as the minHeap property is already taken care of.

`insert(18)` : Node `18` gets added as the left child of `23` and this disturbs the minHeap property — *The child nodes should be less than the parent node*. Now, we go up the chain and find a suitable place for node `18`. We start by comparing it with its parent node `23` and since `18` is less than `23`, these two nodes are swapped. Now we compare `18` with `10`. `18` is greater than `10` which means it is at the correct position in the heap.

And that's how we insert a new node in a Binary Heap! Let's write some code to implement the insert function:

```
insert (node) {

    /* Inserting the new node at the end of the heap array */
    this.heap.push(node)

    /* Finding the correct position for the new node */

    if (this.heap.length > 1) {
        let current = this.heap.length - 1

        /* Traversing up the parent node until the current node (current) is greater th
        while (current > 1 && this.heap[Math.floor(current/2)] > this.heap[current]) {

            /* Swapping the two nodes by using the ES6 destructuring syntax*/
            [this.heap[Math.floor(current/2)], this.heap[current]] = [this.heap[current
            current = Math.floor(current/2)
        }
    }
}
```

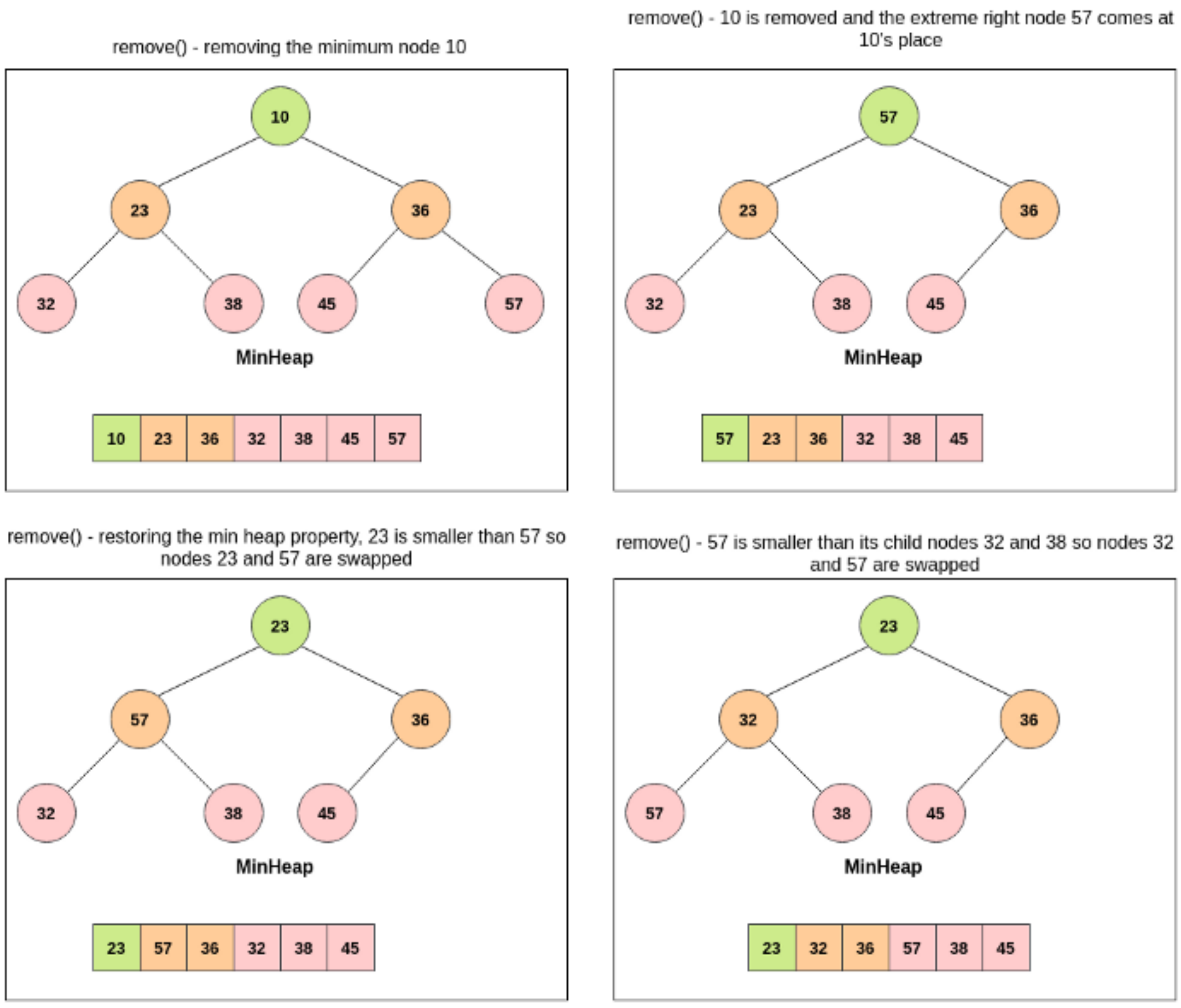
min-heap-insert.js hosted with ❤ by **GitHub** **view raw**

The above code shouldn't be difficult to understand! We start by adding the new node at the end of the array (Remember, a heap is a complete tree except for the last level and it gets filled from left to right).

And now we keep checking the current element with that of its parent. The current node and the parent node are swapped if the current node is smaller than its parent. Please note: the index of the current node is `current` and that of its parent is `current/2`. We're using *ES6 Array destructuring syntax* to swap these two elements. The process of balancing the heap after inserting or removing an element is called *heapify*. When we traverse up the heap to find a suitable place for the new node, it is commonly called *heapifyUp*.

The time complexity of inserting a new element in a binary heap is `O(log n)` where `n` is the number of elements. The number of elements to compare gets reduced to half on every iteration and hence `log n`.

Now let's see what happens when we remove an element from the heap:


remove() - removing the minimum node 10
10
23
36
32
38
45
57
MinHeap
10 23 36 32 38 45 57
remove() - 10 is removed and the extreme right node 57 comes at 10's place
57
23
36
32
38
45
MinHeap
57 23 36 32 38 45
remove() - restoring the min heap property, 23 is smaller than 57 so nodes 23 and 57 are swapped
23
57
36
32
38
45
MinHeap
23 57 36 32 38 45
remove() - 57 is smaller than its child nodes 32 and 38 so nodes 32 and 57 are swapped
23
32
36
57
38
45
MinHeap
23 32 36 57 38 45

Removing the minimum element from a binary heap

Correction in the above image: The headline for the fourth block should be *remove() — 57 is greater than its child nodes 32 and 38 and so nodes 32 and 57 are swapped.*

Here, we're removing the minimum node `10` from the binary heap. When the root node is removed, the extreme right node (`57`) comes at the position of the root node. You can see this in the second illustration that the node `57` becomes the root node. We'll have to restore the minHeap property.

We start traversing down the heap and check if the child nodes are smaller than the parent. If any of the child nodes is smaller than the parent, we swap the parent with the smallest child node.

Please note in the 3rd illustration, we’ve swapped `23` and `57` while in the 4th illustration, we’ve swapped `32` and `57`. The 4th illustration is now a valid minHeap!

Let’s implement the remove function. It’s a bit difficult! Please go through the illustrations once and understand who is getting shifted where and why!

Here’s the code for the remove function:

```
remove() {
    /* Smallest element is at the index 1 in the heap array */
    let smallest = this.heap[1]

    /* When there are more than two elements in the array, we put the right most elemen
        and start comparing nodes with the child nodes
    */
    if (this.heap.length > 2) {
        this.heap[1] = this.heap[this.heap.length-1]
        this.heap.splice(this.heap.length - 1)

        if (this.heap.length === 3) {
            if (this.heap[1] > this.heap[2]) {
                [this.heap[1], this.heap[2]] = [this.heap[2], this.heap[1]]
            }
            return smallest
        }

        let current = 1
        let leftChildIndex = current * 2
        let rightChildIndex = current * 2 + 1

        while (this.heap[leftChildIndex] &&
                this.heap[rightChildIndex] &&
                (this.heap[current] > this.heap[leftChildIndex] ||
                    this.heap[current] > this.heap[rightChildIndex])) {
            if (this.heap[leftChildIndex] < this.heap[rightChildIndex]) {
                [this.heap[current], this.heap[leftChildIndex]] = [this.heap[leftChildI
                current = leftChildIndex
            } else {
                [this.heap[current], this.heap[rightChildIndex]] = [this.heap[rightChil
                current = rightChildIndex
            }

            leftChildIndex = current * 2
            rightChildIndex = current * 2 + 1
        }
    }

    if (this.heap[rightChildIndex] === undefined && this.heap[leftChildIndex] < this.he
        [this.heap[current], this.heap[leftChildIndex]] = [this.heap[leftChildIndex], t
    }

    /* If there are only two elements in the array, we directly splice out the first el
```

We first store the minimum element at index `1` in the variable `smallest`. We'll have to check for the minHeap property if the size of the heap is greater than `2`. If the size of the heap is `2`, we don't have to check anything. We'll simply remove the element at index `1` using the `splice` function. You can see this bit in the `else if` block.

Now, let's get onto the humongous `if` block which is doing the majority of the work. We first put the last element at index `1` and then remove the last element from the heap as:

```
this.heap[1] = this.heap[this.heap.length-1]
this.heap.splice(this.heap.length - 1)
```

It is easy to retain the heap property if there are only `3` elements remaining in the heap. We simply swap the smallest element with the root node. That is it!

If there are more than 3 elements, we traverse down the heap to find a suitable place for the root node. The process of traversing down the heap is commonly called *heapifyDown*.

The condition in the while block looks big but it is not doing much! It simply checks if the current element is smaller than both of its child nodes. Then the smallest child node and the parent node are swapped and the `current` variable is changed accordingly.

The values of `leftChildIndex` and `rightChildIndex` are also changed as:

```
leftChildIndex = current * 2 // i* 2
rightChildIndex = current * 2 + 1 // i * 2 + 1
```

The `remove` method should look easy now! I suggest writing the entire code on your own to have a firm understanding of how operations like `insert` and `remove` are implemented in binary heaps.

Here's the complete code for implementing MinHeap:

Programming JavaScript Data Structures Interview Coding

```
class MinHeap {

    constructor () {
        /* Initialing the array heap and adding a dummy element at index 0 */
        this.heap = [null]
    }

    getMin () {
        /* Accessing the min element at index 1 in the heap array */
        return this.heap[1]
    }

    insert (node) {

        /* Inserting the new node at the end of the heap array */
        this.heap.push(node)

        /* Finding the correct position for the new node */

        if (this.heap.length > 1) {
            let current = this.heap.length - 1

            /* Traversing up the parent node until the current node (current) is greater than
            while (current > 1 && this.heap[Math.floor(current/2)] > this.heap[current]) {

                /* Swapping the two nodes by using the ES6 destructuring syntax*/
                [this.heap[Math.floor(current/2)], this.heap[current]] = [this.heap[current],
                current = Math.floor(current/2)
            }
        }
    }

    remove() {
        /* Smallest element is at the index 1 in the heap array */
        let smallest = this.heap[1]

        /* When there are more than two elements in the array, we put the right most element
            and start comparing nodes with the child nodes
        */
        if (this.heap.length > 2) {
            this.heap[1] = this.heap[this.heap.length-1]
            this.heap.splice(this.heap.length - 1)

            if (this.heap.length === 3) {
```

Try writing the code for MaxHeap on your own. It should not be difficult, you just have to twe few `if` conditions.

Wrapping Up

We learned that the Heaps data structure is an almost complete tree that satisfies the heap property. We can access the min/max elements in `O(1)` from the heap. We can implement Priority Queue using heaps which are essentially used for accessing the elements on a priority basis. In the next article, we'll look at some of the popular interview questions on heaps.

Learn More

Let Everyone In Your Company Share Your Reusable Components

Share your existing technology in a visual way to help R&D, Product, Marketing and everyone else build together.

blog.bitsrc.io

Increase Code Reuse. Reduce Overhead.

How to reduce duplicate code, and scale code-reuse with OSS platform Bit.

blog.bitsrc.io

How to Avoid Duplicate Code with Git+Bit

Leveraging a Git+Bit+NPM workflow to easily scale code sharing.

blog.bitsrc.io