Heapify I

8 min

We've retrieved the minimum element but left our MinHeap in disarray. There's no reason to get discouraged; we've handled this type of problem before, and we can get our MinHeap back in shape!

We'll define a

Preview: Docs Loading link description

method

, .heapify(), which performs a similar role to .bubbleUp(), except now we're moving down the tree instead of up. The current element is a parent that can have either a left child or two children, but not just a right child.

We have written a helper method, .canSwap(), to return true if swapping can occur for either child and false otherwise:

```
canSwap(current, leftChild, rightChild) {
  // Check that one of the possible swap conditions exists
  return (this.exists(leftChild) && this.heap[current] > this.heap[leftChild]
  || this.exists(rightChild) && this.heap[current] > this.heap[rightChild]
    );
  }
```

To maintain the min-heap condition, the parent value has to be less than both its child values. To see if a swap is necessary, starting with the left child, we first check that the child exists and then whether the min-heap condition is broken, i.e. the current element has a value greater than that child's value. If the left child does not break the min-heap condition, the same check is performed on the right child.

Instructions

1. Checkpoint 1 Passed

1.

Define an empty .heapify() method below .bubbleUp() in MinHeap.

2. Checkpoint 2 Passed

2.

We are going to heapify beginning from the index that always points to the minimum value. Declare a let current which points to index 1.

At this stage, index 1 is pointing to the out-of-place value we swapped in while removing the minimum.

3. Checkpoint 3 Passed

3.

We are going to use .canSwap() as we traverse each element in our heap tree.
Since .canSwap() takes 3 arguments: current index, left child index, and right child index, we need to add two more local variables.

- Declare two local variables leftChild and rightChild and assign them to their appropriate values.
- Write a while loop that calls .canSwap()
- At the bottom of the while loop, update the leftChild and rightChild to their appropriate values so that the loop will not run infinitely.

In later exercises, we will continue filling the while loop to restore the heap.

Hint

```
heapify() {
  // assign the current index to 1
  // assign the left child index using a helper method
  // assign the right child index using a helper method
  // call canSwap with three indices in a while loop
  // update the left child index and right child index inside the loop
}
```

MinHeap.js

```
class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }


  popMin() {

    if (this.size === 0) {

      return null

    }

    console.log(`\n.. Swap ${this.heap[1]} with last element ${this.heap[this.size]}`);
```

```
  this.swap(1, this.size);
  const min = this.heap.pop();
  this.size--;
  console.log(`.. Removed ${min} from heap`);
  console.log('..',this.heap);
  return min;
 }

 add(value) {
  console.log(`.. adding ${value}`);
  this.heap.push(value);
  this.size++;
  this.bubbleUp();
  console.log(`added ${value} to heap`, this.heap);
 }

 bubbleUp() {
  let current = this.size;
  while (current > 1 && this.heap[getParent(current)] > this.heap[current]) {
    console.log(`.. swap ${this.heap[current]} with parent ${this.heap[getParent(current)]}`);
    this.swap(current, getParent(current));
    console.log('..', this.heap);
    current = getParent(current);
  }
 }

 heapify() {
  let current = 1;
  let leftChild = getLeft(current);
```

```
    let rightChild = getRight(current);

    while (this.canSwap(current, leftChild, rightChild)) {
      leftChild = getLeft(current);
      rightChild = getRight(current);
    }
  }

  exists(index) {
    return index <= this.size;
  }

  canSwap(current, leftChild, rightChild) {
    // Check that one of the possible swap conditions exists
    return (
      this.exists(leftChild) && this.heap[current] > this.heap[leftChild]
      || this.exists(rightChild) && this.heap[current] > this.heap[rightChild]
    );
  }

  swap(a, b) {
    [this.heap[a], this.heap[b]] = [this.heap[b], this.heap[a]];
  }

}

const getParent = current => Math.floor((current / 2));
const getLeft = current => current * 2;
const getRight = current => current * 2 + 1;
```

```
module.exports = MinHeap;
```

script.js

```
// import MinHeap class
const MinHeap = require('./MinHeap');

// instantiate a MinHeap class
const minHeap = new MinHeap();

// helper function to return a random integer
function randomize() { return Math.floor(Math.random() * 40); }

// populate minHeap with random numbers
for (let i=0; i < 6; i++) {
  minHeap.add(randomize());
}

// display the bubbled up numbers in the heap
console.log('Bubbled Up', minHeap.heap);

// remove the minimum value from heap
minHeap.popMin();
```