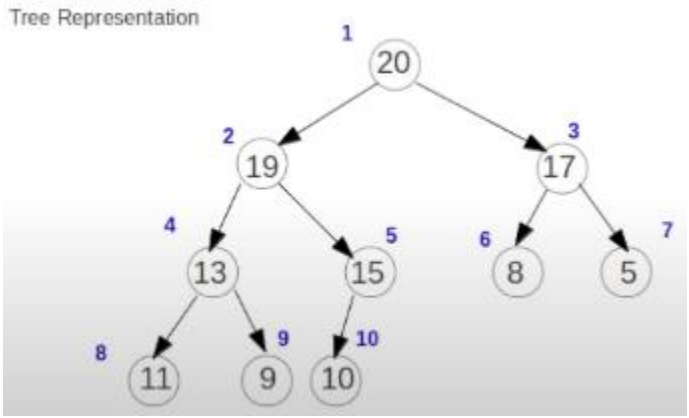
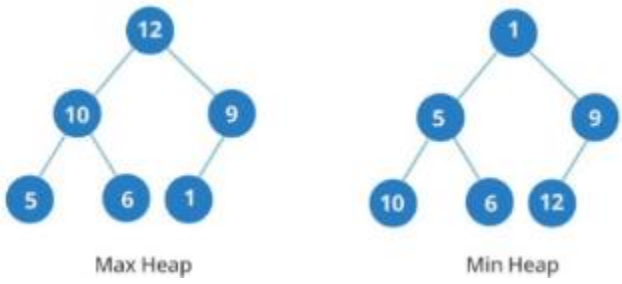
Heap Data Structure

Heaps are complete binary trees, ie, all levels are filled... if it is partially filled, it starts filling from left to right:

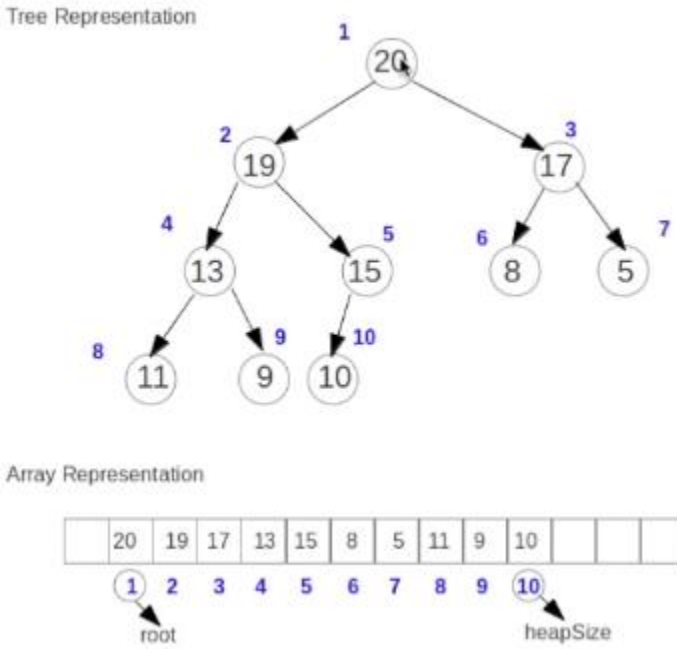


Max Heap: parent nodes are bigger than their respective child nodes.

Min Heap: parent nodes are smaller than their respective child nodes.



… Heaps may also be implemented in arrays, not just trees:



… How do we make sense of this? Let's take a look at these equations:

```
// left child: i * 2
// right child: i * 2 + 1
// parent: i / 2
```

Index 1 = 1

Index 2 = 1*2 = 2

Index 3 = 1*2+1 = 3

Index 4 = 2*2 = 4

Index 5 = 2*2+1 = 5

Index 6 = 3*2 = 6

Index 8 = 4*2 = 8

Index 10 = 5*2 = 10

Index 7 = 3*2+1 = 7

Index 9 = 4*2+1 = 9

**Notice how there is no index of zero... this is to ensure the math works out in the equations.

Step One: Inserting and Removing elements:

We create function **minHeap** and null array **heap** ...
... null means indexed at zero.

Insert:

```
let MinHeap = function() {

  let heap = [null];

  this.insert = function(num) {
    heap.push(num);
    if (heap.length > 2) {
      let idx = heap.length - 1;
      while (heap[idx] < heap[Math.floor(idx/2)]) {
        if (idx >= 1) {
          [heap[Math.floor(idx/2)], heap[idx]] = [heap[idx], heap[Math.floor(idx/2)]];
          if (Math.floor(idx/2) > 1) {
            idx = Math.floor(idx/2);
          } else {
            break;
          };
        };
      };
    };
  };
};
```

When we enter a number, we push it to the end of the heap.

If the length of the heap is less than 2, that means there is one or zero items in the heap.

If the length of the heap is more than 2, that means there is more than one item in the heap...
... which means we need to find the last index, while that heap is less than its parent... which means we need to move it up over its parent.

If the parent node is not the root node, we swap until the smaller node is at the top... once there is no need to swap, we break out of the while loop.

Note: the statement containing [heap[idx]] is the destructuring syntax where the swapping occurs.

Remove:

```
this.remove = function() {
  let smallest = heap[1];
  if (heap.length > 2) {
    heap[1] = heap[heap.length - 1];
    heap.splice(heap.length - 1);
    if (heap.length == 3) {
      if (heap[1] > heap[2]) {
        [heap[1], heap[2]] = [heap[2], heap[1]];
      };
      return smallest;
    };
    let i = 1;
    let left = 2 * i;
    let right = 2 * i + 1;
    while (heap[i] >= heap[left] || heap[i] >= heap[right]) {
      if (heap[left] < heap[right]) {
        [heap[i], heap[left]] = [heap[left], heap[i]];
        i = 2 * i
      } else {
        [heap[i], heap[right]] = [heap[right], heap[i]];
        i = 2 * i + 1;
      };
      left = 2 * i;
      right = 2 * i + 1;
        if (heap[left] == undefined || heap[right] == undefined) {
          break;
        };
      };
    } else if (heap.length == 2) {
      heap.splice(1, 1);
    } else {
      return null;
    };
    return smallest;
  };

  this.sort = function() {
    let result = new Array();
    while (heap.length > 1) {
      result.push(this.remove());
    };
    return result;
  };

};
```

We declare that the smallest is the first index... seems simple enough, but if we remove it, how do we rearrange the whole structure? Which element replaces it? And will that act of replacement set off a chain of other replacements and thus alter the whole heap structure?

If we have more than one node, we set the smaller one to be the first node and remove the larger node.

If there were three nodes, we swap between the smallest node and second smaller node to allow for the smaller node to be set as the first node.

Next, we use the equations that were explained in the previous page: While the root node is greater than or equal to its left child and right child, we will move swap the nodes until everything is set accordingly.