Priority Queue

Created	@September 19, 2022 9:00 PM
Class	
Type	
Materials	
Reviewed	☐

We must have already heard about normal simple queues. In Simple queues, we follow `FIFO`. The element which was inserted first is given priority over other elements. But, what if we want to change the criteria of priority to something else ? We want a structure, that can work like a queue, but we can set custom priority logic, for example, let's say doesn't matter when the element was inserted, but if we want to remove we will remove the largest element inserted uptill now.

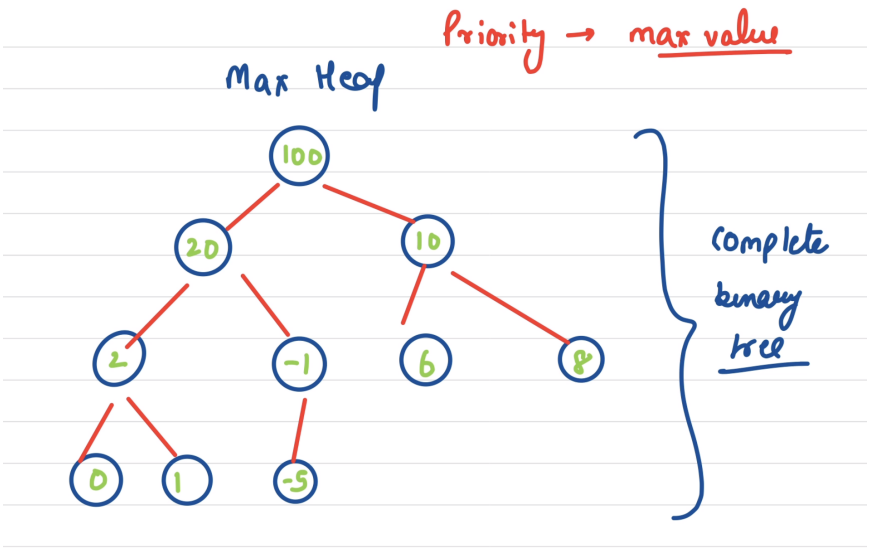
So can we implement this structure using arrays ? If we will use arrays, whenever we remove an element we have to sort the array again based on the priority order. And when we insert then also we need to insert based on priority order, for that we need to do comparisons with all elements. This is not efficient.

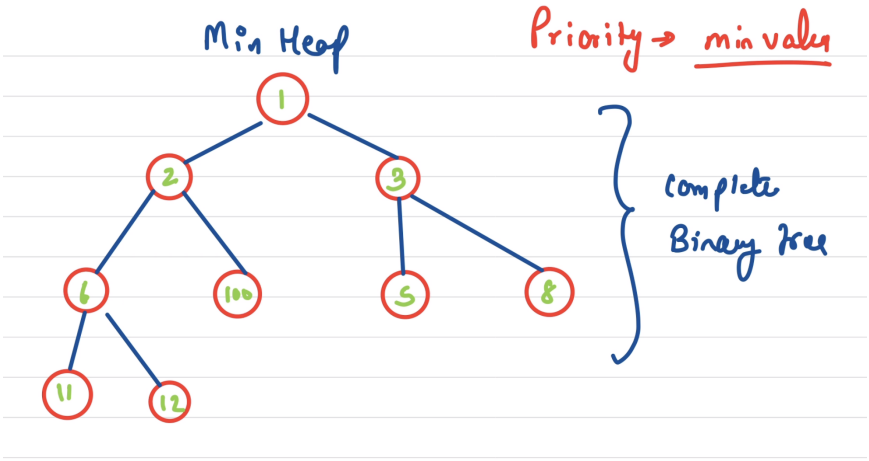
Stacks and linked lists are also not going to help.

So for our rescue we have a new data structure that can help us to implement priority queue which is called as `Heaps`

Heaps

- Heaps posses a structure of a complete binary tree.
- Having a tree like structure means there is a parent child relationship, so for heaps, priority of parent is always higher than priority of child.

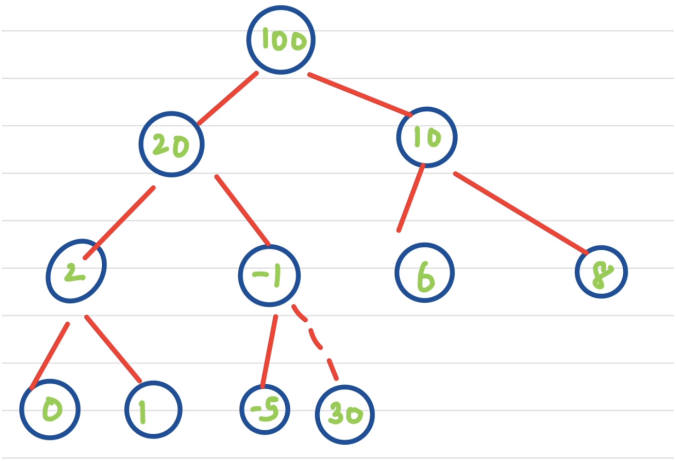




The priority can be any custom mathematical equation. If we use min value as higher prirority then that type of heap is called `Min heap` and if we use max value as higher priority then that type of heap is called `Max heap`.

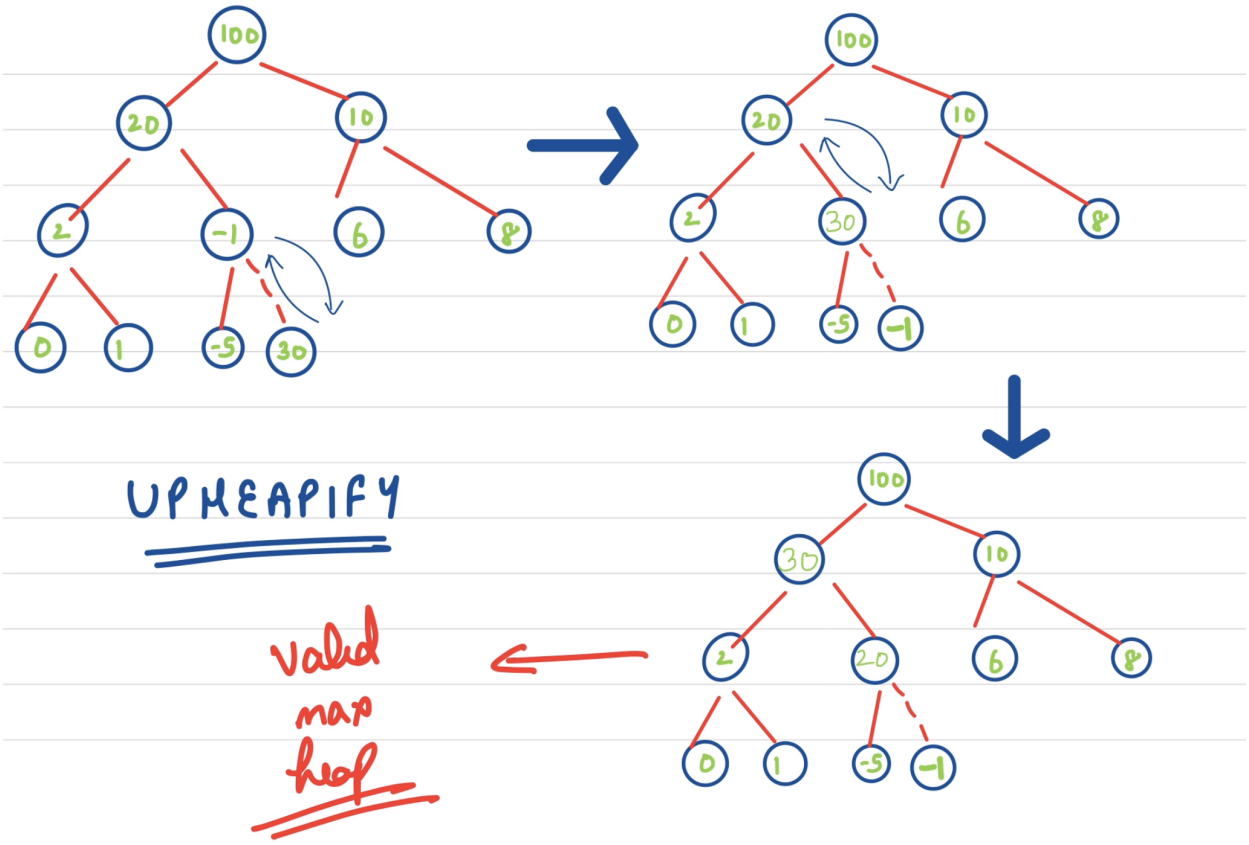
Upheapification

Now we know that heap is a complete binary tree, so if we add a new value it will be added as a next available leaf node. But due to this addition there can be a case that some subtrees might loose the property of heap.



In the above max heap, we added 30, it is still a complete binary tree but the subtree of -1 and 20 are no more a valid max heap.

In order to resolve this what we can do is, starting from the node we added, we can compare it to it's parent and if the value of child is greater than of parent, we can swap the values, till the time we get a valid heap



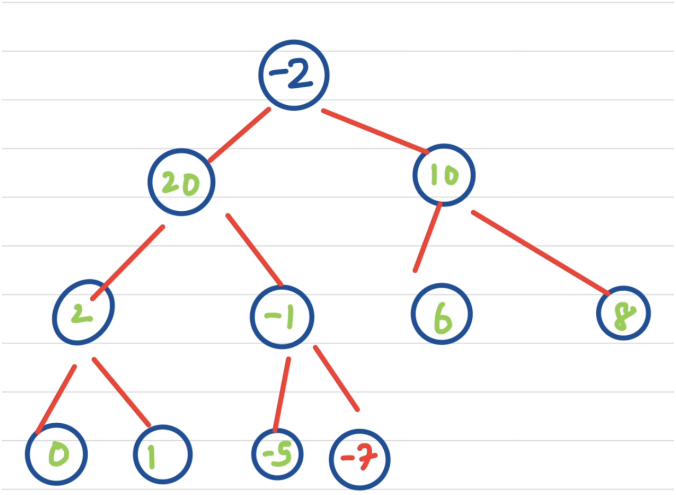
In the above tree, we first swapped 30 with -1, and then 30 with 20 and made the whole tree a valid max heap.

This process is called `Upheapification`.

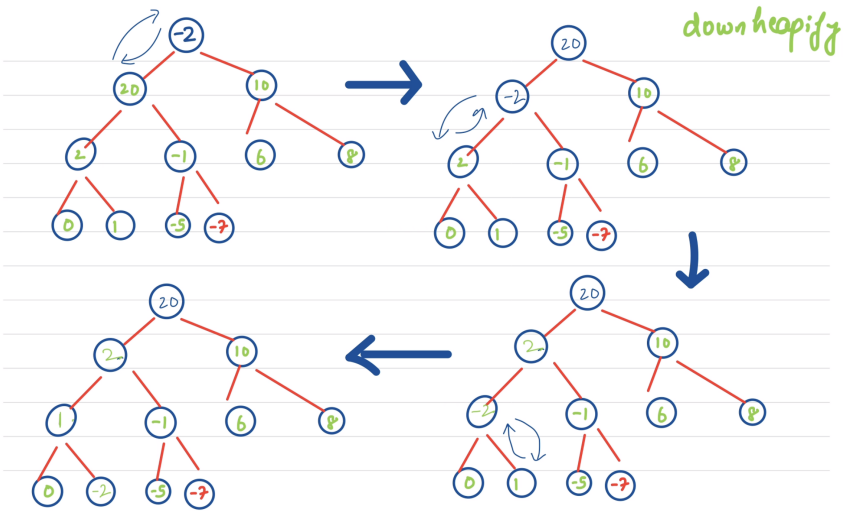
We can use this operation to implement insert operation in a heap later.

Downheapification

Assume w.r.t root, the left subtree is a max heap, right subtree is a max heap, but the whole tree is not a max heap due to wrong value on root. How to make the whole tree max heap.



The above tree has a root, -2. The LST of -2 is a max heap, RST of -2 is a max heap but whole tree is not a max heap. In order to make it a max heap, w.r.t every invalid subtree, compare the root with the max of the root and children, and then swap the child with the root, till the time we get a valid heap.



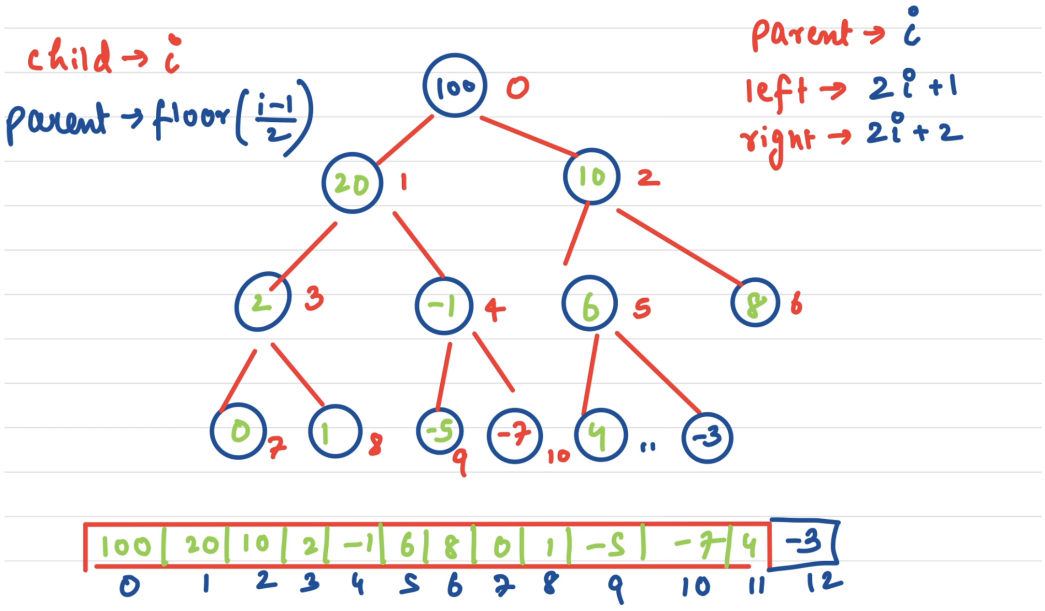
This process is called `downheapification`, and can be useful to remove an element from the heap.

How heaps are implemented ?

As heaps are trees, we can implement them using node class. But this will be tedious and because heaps are Complete Binary trees we can implement it using arrays also.

We will be just storing an array, that will represent a tree like structure. So heap as a data structure is hierarchical in nature, but is implemented in a linear Data structure.

Because we have a complete binary tree, whenever we add a node we can add at the last of the array, we don’t need to care about spacing.



Now the problem is if we have array how can we go from parent to child. This is very easily solvable, if the parent is present at the ith index of the array, the left child will be at `2i+1` and right child will be at `2i+2`. Also if the child is at ith index, it's parent will be at `floor((i-1)/2)`

```
class MaxHeap {
    constructor() {
        this.arr = [];
    }
    upheapify(idx) {
        /**
         * Time: O(logn) Space: O(1)
         */
        while(idx > 0) { // we cannot upheapify root, we will stop the loop, when we reach
root
            let parentIdx = Math.floor((idx - 1) / 2);
            if(this.arr[parentIdx] < this.arr[idx]) {
                // if parent is less than child, swap them
                let temp = this.arr[parentIdx];
                this.arr[parentIdx] = this.arr[idx];
                this.arr[idx] = temp;

                // move upwards
                idx = parentIdx;
            } else {
                // alread a max heap
                break;
            }
```

```
        }
    }
    insert(x) {
        // this function inserts x correctly in max heap
        this.arr.push(x);
        this.upheapify(this.arr.length - 1); // arr.len - 1 is the index on which x is add
ed
    }
}
```

To remove the root, we can just simply, swap the root with last index element, remove the last index element and do downheapify.

```
class MaxHeap {
    constructor() {
        this.arr = [];
    }
    upheapify(idx) {
        /**
         * Time: O(logn) Space: O(1)
         */
        while(idx > 0) { // we cannot upheapify root, we will stop the loop, when we reach
root
            let parentIdx = Math.floor((idx - 1) / 2);
            if(this.arr[parentIdx] < this.arr[idx]) {
                // if parent is less than child, swap them
                let temp = this.arr[parentIdx];
                this.arr[parentIdx] = this.arr[idx];
                this.arr[idx] = temp;

                // move upwards
                idx = parentIdx;
            } else {
                // alread a max heap
                break;
            }
        }
    }

    downHeapify(idx) {
        /**
         * Time: O(logn)
         * Space: O(1)
         */
        while(idx < this.arr.length) {
            let left = 2*idx + 1;
            let right = 2*idx + 2;
            let greatest = idx; // initially assume root is the greatest
            if(left < this.arr.length && this.arr[left] > this.arr[greatest]) {
                // if left child exist and it is greater than root, then greatest is left
```

```
                greatest = left;
            }
            if(right < this.arr.length && this.arr[right] > this.arr[greatest]) {
                // if right child exist and right is greater than max(root, left) then rig
ht is greatest
                greatest = right;
            }
            if(greatest == idx) {
                // we dont need to swap and we can stop
                break;
            }
            // swap
            let temp = this.arr[greatest];
            this.arr[greatest] = this.arr[idx];
            this.arr[idx] = temp;
            idx = greatest;
        }
    }

    removeRoot() {
        // swap the root node with last index value
        let temp = this.arr[0];
        this.arr[0] = this.arr[this.arr.length - 1];
        this.arr[this.arr.length - 1] = temp;

        this.arr.pop(); // remove the last index element

        this.downHeapify(0);
    }

    insert(x) {
        // this function inserts x correctly in max heap
        this.arr.push(x);
        this.upheapify(this.arr.length - 1); // arr.len - 1 is the index on which x is add
ed
    }

    display() {
        console.log(this.arr);
    }
}

let hp = new MaxHeap();
hp.insert(4);
hp.insert(9);
hp.insert(-1);
hp.insert(10);
hp.insert(30);
hp.insert(15);
hp.insert(12);
hp.display();
hp.removeRoot();
hp.display();
```

Note: if we want to delete any element present at some ith index, we can replace the element with +inf and upheapify so that inf goes to the root and then delete the root.

Given an array of n elements, create a heap using this array.