**Binary Heap (TREE Data Structure) &  
 Priority Queue**

Heaps are another category of Trees. Where Binary Heap are one of the type of Heaps category.

**WHAT IS A BINARY HEAP?**

Very similar to a binary search tree, but with some different rules!

In a **MaxBinaryHeap**, parent nodes are always larger than child nodes. In a **MinBinaryHeap**, parent nodes are always smaller than child nodes.

**Note:**

* In Binary Heap we always fill up the Left side first then Right Side.
* Binary heap, which is a type of a Heap, which itself is a type of a Tree.

**MAX BINARY HEAP**

* Each parent has at most two child nodes
* The value of each parent node is always greater than its child nodes
* In a max Binary Heap the parent is greater than the children, but there are no guarantees between sibling nodes.
* A binary heap is as compact as possible. All the children of each node are as full as they can be and left children are filled out first

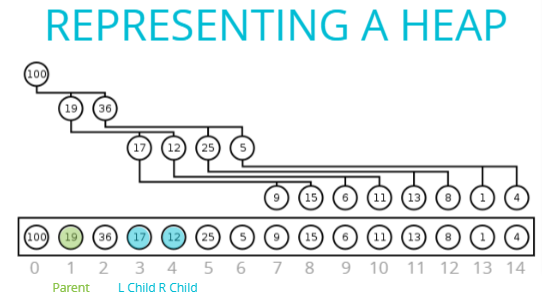
**Why do we need to know this?**

Binary Heaps are used to implement Priority Queues, which are very commonly used data structures.

They are also used quite a bit, with graph traversal algorithms.

**AN EASY WAY OF STORING A BINARY HEAP**

**A LIST/ARRAY**



What If we’ve **Parent Node** and want to find its **Child**. Follow the following formula.

For **Left Node** of any Parent:

**2n + 1**where, **n** is the index of the parent node.

For **Right Node** of any Parent:

**2n + 2**where, **n** is the index of the parent node.

EX: Find L-Child & R-Child of Node 19.  
 n = 1  
 L-Child = 2n+1 = 2\*1+1 = 3  
 R-Child = 2n+2 = 2\*1+2 = 4  
At index-3, Node is 17 (L-Child Node)  
At index-4, Node is 12 (R-Child Node)

What If we’ve **Child Node** and want to find its **Parent**. Follow the following formula.

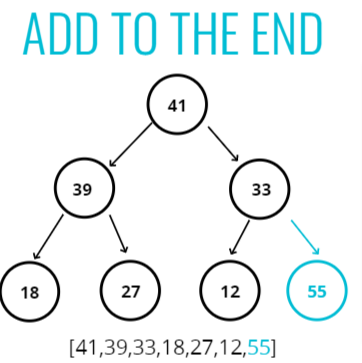
For **Parent Node** of any Child:

**Math.floor((n – 1) / 2)**where, **n** is the index of the Child node.

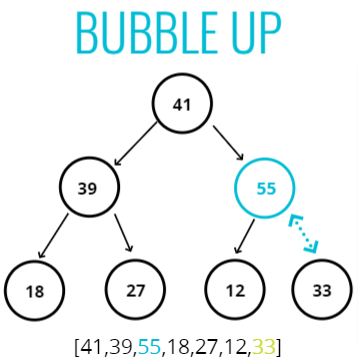
EX: Find Parent Node of Node-17 & Node-12.  
 n = 3 (Node-17) & n = 4 (Node-13)  
 Node-17 = Math.floor((n-1)/2) = Math.floor((3-1)/2) = 1 Node-17 = Math.floor((n-1)/2) = Math.floor((4-1)/2) = 1  
  
At index-1, Node is 19 (Parent Node)

**Adding/Inserting to a MaxBinaryHeap:**

* Add to the end



* Bubble up (Process of Swapping child with the parent)



**Visualization:**

Available in Code:

**Approach**

* Keep in mind,
  + MaxBinaryHeap means Parent node always be Greater than Child node.
  + Inserting an element in maxBinaryHeap tree, it will begin by adding as the last node (leaf node) of the tree.
  + We're going arrange MaxBinaryHeap tree in Array Data Structure. Means, New element will be added at the beginning of array.
  + In array, To find parent of any Inserted node, we'll use floored value of (n-1)/2. It will be the index of parent element.
* Push new Element to the array.
* Start bubblingUp the Element with its above parent in the tree.
* Store index of new Element in 'idx' and Store value of new Element in 'element' variable.
* Start a Loop till 'idx' of inserted Element =< 0
  + Store index of Parent Element of inserted element in 'parentIdx' and Store value of parent Element in 'parent' variable.
  + Check, if (inserted 'element' is smaller than its 'parent') break the loop, return the array.
  + Otherwise, Start Swapping:
  + At index of Inserted Element <-- Set Parent Element
  + At index of Parent Element <-- Set Inserted Element
  + Assign 'parentIdx' to the 'idx' of Inserted Element (As Inserted element moving Up along the tree, started acquiring index of parent Element, which lead in the decrement of it 'idx')

**INSERT PSEUDOCODE in MaxBinaryHeap**

* Push the value into the values property on the heap
* Bubble Up:
* Create a variable called index which is the length of the values property – 1
* Create a variable called parentIndex which is the floor of (index-1)/2
* Keep looping as long as the values element at the parentIndex is less than the values element at the child index
* Swap the value of the values element at the parentIndex with the value of the element property at the child index
* Set the index to be the parentIndex, and start over!

**INSERT CODE for MaxBinaryHeap**

class MaxBinaryHeap {

constructor() {

this.value = [41, 39, 33, 18, 27, 12];

}

insert(element){

this.value.push(element);

const showMaxBinaryHeap = this.bubbleUp();

return showMaxBinaryHeap;

}

bubbleUp(){

let idx = this.value.length-1;

const element = this.value[idx];

while(idx>0){

let parentIdx = Math.floor((idx-1)/2);

const parent = this.value[parentIdx];

if(element<=parent) break;

//Start Bubbling Up (Swapping)

this.value[idx] = parent;

this.value[parentIdx] = element;

idx = parentIdx;

}

return this.value;

}

}

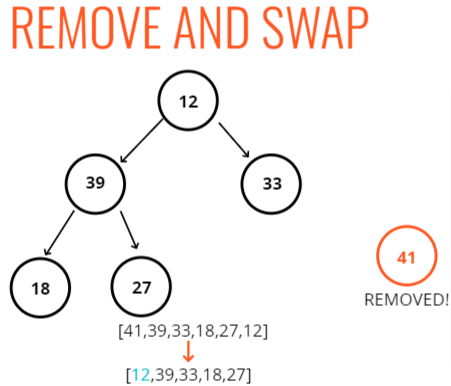
let maxBinaryHeap = new MaxBinaryHeap();

maxBinaryHeap.insert(55);

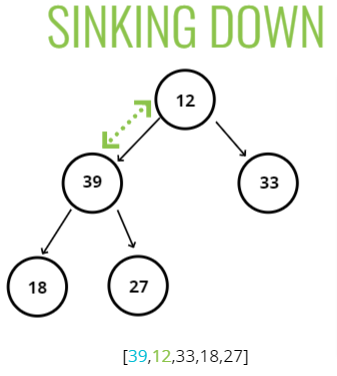
**Output:**[55, 39, 41, 18, 27, 12, 33]

**Removing (Extract Max) to a MaxBinaryHeap:**

* Remove the root (Removing Element/Node of maxium Value)



* Replace with the most recently added
* Adjust (sink down or Bubble Down)  
  The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called down-heap (also known as bubble-down, percolate-down, sift-down, trickle down, heapify-down, cascade-down, and extract-min/max).



**Visualization:**

Available in the Code.

**Approach:**

* Store a maxBinaryHeap into an Array or List
* ExtractMax: In maxBinaryHeap heap Parent is always > Child. So root will always be the highest Element in maxBinaryHeap. This is what we've to Remove or say Extract i.e. why it's called ExtractMax.
* Store root Element to 'max' and Pop out the last element from the array (means, last node of Bineary heap) and store it in 'end' variable.
* If (length of array or say BinaryHeap is Greater than 0 then only we can perform extractMax or Remove Operaition on Maximum Element)
  + Set the End Element at the root of BinaryHeap. (This will cause removal of Old Root or Maximum Element from the Tree.)
    - This is how remove Operaition has complete. Now we've to arrange current Binary Tree in such a way that will look like a Valid MaxBinaryHeap.
* Start sinkingDown or say Bubbling Down.
  + Assign 0 to 'idx' variable. This would be the index of current root Element that moveout from last to top in BinaryHeap.
  + Store that current root Element to an 'element' variable and current length of the array or current depth of BinaryHeap intor 'length' variable.
  + Start a loop.
    - Store index of left Child of 'element' into 'leftChildIdx'(2n+1) & right Child into 'rightChildIdx'(2n+2).
* Declare 'LeftChild' & 'RightChild' variable and Initialize 'swap' variable with null. (this swap will furhter help in terminating the loop).
* store left & right Child of current 'element' into 'LeftChild' & 'RightChild' variable.
* If(leftChildIdx<length of array) //To avoid getting out of boundary in order for searching child of current 'element' accross the array/tree.

if(current element < its LeftChild) --> Assign leftChildIdx to 'swap' variable.

* + - If(rightChildIdx<length of array) //To avoid getting out of boundary in order for searching child of current 'element' accross the array/tree.

if(swap===null && current element < its RightChild || swap!==null && RightChild>LeftChild) --> Assign rightChildIdx to 'swap' variable.

* + - If (swap === null) break out of the loop.
    - Start Swapping of current 'element' with its child

Set current 'element' at the index of its child element's index

Set child Element at the index of its parent element's or say current 'element''s index

* + - Set index of child Element to the index of it's parent Element or say current 'element'.

**REMOVING Pseudo Code to MaxBinaryHeap:**

* Swap the first value in the values property with the last one
* Pop from the values property, so you can return the value at the end.
* Have the new root "sink down" to the correct spot...​
* Your parent index starts at 0 (the root)
* Find the index of the left child: 2 \* index + 1 (make sure its not out of bounds)
* Find the index of the right child: 2\*index + 2 (make sure its not out of bounds)
* If the left or right child is greater than the element...swap. If both left and right children are larger, swap with the largest child.
* The child index you swapped to now becomes the new parent index.
* Keep looping and swapping until neither child is larger than the element.
* Return the old root!

**REMOVE/EXTRACTMAX CODE for MaxBinaryHeap**

class MaxBinaryHeap {

constructor() {

this.value = [41,39,33,18,27,12];

}

extractMax(){

const max = this.value[0],

end = this.value.pop();

if(this.value.length>0){

this.value[0] = end;

this.sinkDown();

return max;

}

}

**//BubblingDown or say sinkDown**

sinkDown(){

let idx = 0,

element = this.value[0],

length = this.value.length;

while(true){

let leftChildIdx = 2\*idx +1,

rightChildIdx = 2\*idx +2,

LeftChild, RightChild,

swap = null;

LeftChild = this.value[leftChildIdx];

RightChild = this.value[rightChildIdx];

if(leftChildIdx<length){

if(element<LeftChild){

swap = leftChildIdx;

}

}

if(rightChildIdx<length){

if(swap===null && RightChild>element || swap!==null && RightChild>LeftChild){

swap = rightChildIdx;

}

}

if(swap===null) break;

this.value[idx] = this.value[swap];

this.value[swap] = element;

idx = swap;

}

}

}

let maxBinaryHeap = new MaxBinaryHeap();

maxBinaryHeap.extractMax();

**Output:**41 **//Removed Max Element from the Heap.**

[39, 27, 33, 18, 12]

Note:

Binary Heap works really well because **insertion** and **removal** are they have a **time complexity of log(n)**.

**Priority Queue**

**WHAT IS A PRIORITY QUEUE?**

A data structure where each element has a priority. Elements with higher priorities are served before elements with lower priorities.

JUST LIKE before, Here we’ll have,

Class: Priority Queue   
And along with Value

But also have,   
Another Class: Node  
And along with Value & priority

Since in priority queue, the lowest priority number has the highest priority. So, We’re going to use Min Binary Heap for Priority Queue. (means most minimum priority number node will stay at root place).

Priority: A lower value is going to be served first.

**Note:**

* When we insert/enqueuer something, we just compare the priority level, we don’t compare the value of the element.
* When we remove something, we’re going to remove the root of the heap, the top level thing, which is the highest, technically the lowest number but the highest priority.

**Priority Queue Pseudo Code:**

* Write a Min Binary Heap - lower number means higher priority.
* Each Node has a val and a priority. Use the priority to build the heap.
* Enqueue method accepts a value and priority, makes a new node, and puts it in the right spot based off of its priority.
* Dequeue method removes root element, returns it, and rearranges heap using priority.

**Priority Queue Code:**

class Node {

constructor(val, priority) {

this.val = val;

this.priority = priority;

}

}

class PriorityQueue {

constructor() {

this.value = [];

}

enqueue(value, priority){

let newNode = new Node(value, priority);

this.value.push(newNode);

this.bubbleUp();

// return this.value;

}

bubbleUp(){

let idx = this.value.length-1,

element = this.value[idx];

while(idx>0){

let parentIdx = Math.ceil((idx-1)/2),

parent = this.value[parentIdx];

if(element.priority >= parent.priority) break;

this.value[parentIdx] = element;

this.value[idx] = parent;

idx = parentIdx;

}

}

dequeue(){

const min = this.value[0],

end = this.value.pop();

if(this.value.length>0){

this.value[0] = end;

this.sinkDown();

}

return min;

}

//BubblingDown or say sinkDown

sinkDown(){

let idx = 0,

element = this.value[0],

length = this.value.length;

while(true){

let leftChildIdx = 2\*idx +1,

rightChildIdx = 2\*idx +2,

LeftChild, RightChild,

swap = null;

if(leftChildIdx<length){

LeftChild = this.value[leftChildIdx];

if(element.priority > LeftChild.priority){

swap = leftChildIdx;

}

}

if(rightChildIdx<length){

RightChild = this.value[rightChildIdx];

if(swap===null && RightChild.priority<element.priority || swap!==null && RightChild.priority<LeftChild.priority){

swap = rightChildIdx;

}

}

if(swap===null) break;

this.value[idx] = this.value[swap];

this.value[swap] = element;

idx = swap;

}

}

}

let priorityQueue = new PriorityQueue();

priorityQueue.enqueue("This",3);

priorityQueue.enqueue("Chandan",5);

priorityQueue.enqueue("There",2);

priorityQueue.enqueue("Hi",1);

priorityQueue.enqueue("is",4);

**Output:**[Node, Node, Node, Node, Node]  
0: Node {val: 'Hi', priority: 1}  
1: Node {val: 'There', priority: 2}  
2: Node {val: 'is', priority: 4}  
3: Node {val: 'This', priority: 3}  
4: Node {val: 'Chandan', priority: 5}

length: 5

**Big O of Binary Heaps**

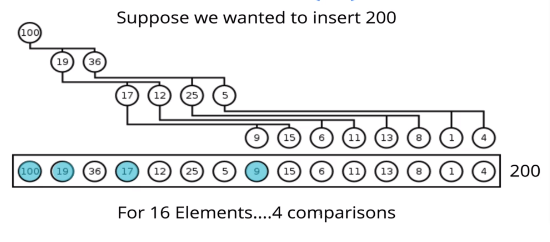
Insertion - **O(log N)**

Removal - **O(log N)**

Search - **O(N)**

**O(logN)** means log2(N). Order of logN base2.

**Why log(N)?**



2? COMPARISON = 16 Elements

24 COMPARISON = 16 Elements

Comparisons = Operations

**RECAP**

* Binary Heaps are very useful data structures for sorting, and implementing other data structures like priority queues
* Binary Heaps are either MaxBinaryHeaps or MinBinaryHeaps with parents either being smaller or larger than their children.
* With just a little bit of math, we can represent heaps using arrays!