**Binary Heaps**

*Definition:*

* Very similar to search tree
* In a MaxBinaryHeap, parent nodes are always larger than child nodes
* In a MinBinaryHeap, parent nodes area always smaller than child nodes
* A binary heap is a as compact as possible. All the children of each node are as full as they can be and left children are filled out first

*Storage Index Calculation:*

* For every index of an array n
  + The left child is stored at 2n+1
  + The right child is stored at 2n+2
* For any child node at index n
  + Its parent is at index Math.floor((n-1)/2)

*Pseudocode for Binary Heap:*

insert:

* 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

extractMax (extract the root then replace the root with the last element, then sink down):

* 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 is not our of bounds)
  + If the left or right child is larger than the element…swap. If both left and right child are larger, swap with the greater one
  + 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

*Solutions for Binary Heap:*

class MaxBinaryHeap() {  
 constructor() {

this.val = [];

}

insert(element) {

this.val.push(element);

this.bubbleUp();

}

bubbleUp() {

var index = this.val.length – 1;

const element = this.val[index];

while(index > 0) {

var parentIndex = Math.floor((index – 1)/2);

var parent = this.val[parentIndex];

if (element <= parent) break;

this.val[parentIndex] = element;

this.val[index] = parent;

index = parentIndex;

}

}

extractMax() {

const max = this.val[0];

const end = this.val.pop();

// replace last item to be the parent  
 if (this.val.length > 0) {

this.val[0] = end;

this.sinkDown();

}

return max;

}

sinkDown() {

var index = 0;

const length = this.val.length;

const element = this.val[0];

while(true) {

var leftChildIndex = 2 \* index + 1;

var rightChildIndex = 2 \* index + 2;

var leftChild, rightChild;

var swap = null;

if (leftChild < length) {

leftChild = this.val[leftChildIndex];

if (leftChild > element) {

swap = leftChildIndex;

}

}

if (rightChild < length) {

rightChild = this.val[rightChildIndex];

if (swap === null && rightChild > element) ||   
 (swap !== null && rightChild > leftChild){

swap = rightChildIndex;

}

}

if (swap === null) break;

this.val[index] = this.val[swap];

this.val[swa] = element;

index = swap;

}

}

}

**Priority Queue**

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

*Pseudocode for Priority Queue:*

* Write a Min Binary Heap – lower number means higher priority
* Each node has a value and a priority. Use the priority to build the heap
* Enqueue method accepts a value and a 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 heaps using priority.

*Solutions for Priority Queue:*

class Node{

constructor (val, priority) {

this.val = val;

this.priority = priority;

}

}

class PriorityQueue() {  
 constructor() {

this.val = [];

}

enqueue(val, priority) {

var newNode = new Node(val, priority);

this.val.push(newNode);

this.bubbleUp();

}

bubbleUp() {

var index = this.val.length – 1;

const element = this.val[index];

while(index > 0) {

var parentIndex = Math.floor((index – 1)/2);

var parent = this.val[parentIndex];

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

this.val[parentIndex] = element;

this.val[index] = parent;

index = parentIndex;

}

}

dequeue() {

const min = this.val[0];

const end = this.val.pop();

// replace last item to be the parent  
 if (this.val.length > 0) {

this.val[0] = end;

this.sinkDown();

}

return max;

}

sinkDown() {

var index = 0;

const length = this.val.length;

const element = this.val[0];

while(true) {

var leftChildIndex = 2 \* index + 1;

var rightChildIndex = 2 \* index + 2;

var leftChild, rightChild;

var swap = null;

if (leftChild < length) {

leftChild = this.val[leftChildIndex];

if (leftChild.priority < element.priority) {

swap = leftChildIndex;

}

}

if (rightChild < length) {

rightChild = this.val[rightChildIndex];

if (swap === null && rightChild.priority < element.priority) ||   
 (swap !== null && rightChild.priority < leftChild.priority){

swap = rightChildIndex;

}

}

if (swap === null) break;

this.val[index] = this.val[swap];

this.val[swa] = element;

index = swap;

}

}

}

*Big O:*

Insertion – O(logN)

Removal – O(logN)

Search – O(N)