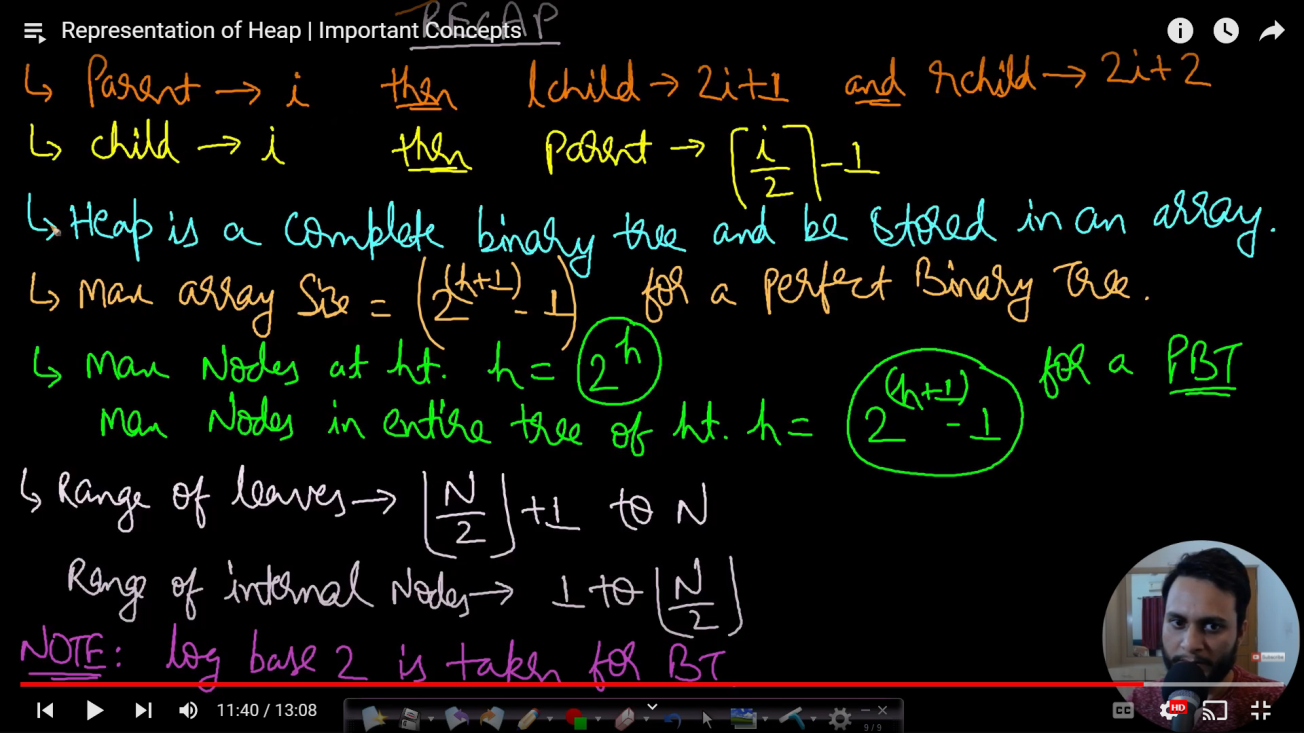
**HEAP**

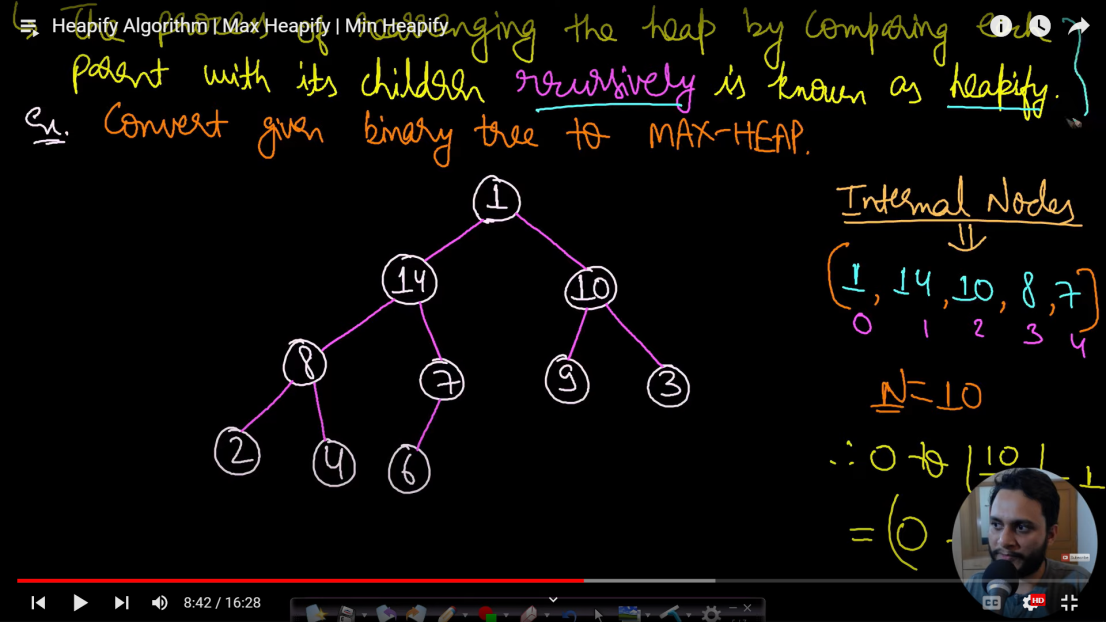
**Heap is always implemented as an ARRAY and not as a B-Tree. We just treat it as a binary tree.**

****

Max/Min Heap = Root will be the maximum/minimum element in the subtree.

Leaf is always a min and max heap.

HEAPIFY



Max Heap

Base Condition

* Reach Leaf
* curr > L && curr > R

For min heap second condition will be reversed

Pseudo algo

MAX\_HEAPFY (arr,i){ // i is the position where the heap is imbalanced

L = 2\*i+1;

R = 2\*i+2;

largest = i;

if ( L < arr.size() and arr[L] > arr[largest] ) // arr[L] < arr[largest] for min heap

largest = L;

if ( R < arr.size() and arr[R] > arr[largest] ) // arr[R] < arr[largest] for min heap

largest = R;

if( largest != i )

swap (arr[i] , arr[largest] );

MAX\_HEAPIFY ( arr,largest ); //recursively calling for the subtree to heapify

}

BUILD HEAP

We can heapify an element at an index “i” if all elements in both left and right subtrees are following heap property.i.e left and right subtree is already a heap.

Given an array, make a heap out of it

IMP POINT = 0 to floor(N/2) are internal nodes and only needed to be heapified as leaf nodes are already heapified.

Procedure = Heapify all internal nodes ( Bottom Up Method )

Pseudo Code

for ( i = (arr.size()/2) ; i>=0 ; i-- ){

MAX\_HEAPIFY (arr, i);

}

EXTRACT MAX / MIN

* Save max/min value.
* Copy last element to root i.e first element of array.
* Decrease the size of array by one.
* Heapify root i.e i=0.

INCREASE KEY MAX HEAP / DECREASE KEY MIN HEAP

* Increase a particular value of node.
* Percolate up till

1. Parent > Current\_Node / Parent < Current\_Node in Min Heap
2. Current\_Node becomes root of the tree.

Pseduo Algo

INCREASE\_KEY ( arr,i,val ){

arr[i] = val;

parent = i/2;

while( i > 0 and arr[i] > arr[parent] ){ / **while( i > 0 and arr[i] <arr[parent] ) in Min Heap**

swap( arr[i] , arr[parent]);

i= parent;

parent = i/2;

}

DECREASE KEY MAX HEAP / INCREASE KEY MIN HEAP

* Decrease value of a particular node.
* Perform MAX\_HEAPIFY / MIN\_HEAPIFY of current node.

INSERT ELEMENT

* Insert Node at the end of array
* Increase size of array.
* Perform INCREASE KEY MAX HEAP / DECREASE KEY MIN HEAP

HEAP SORT

To Sort in Ascending Order, Create a min heap and perform extract min N-1 times.

To Sort in Descending Order, Create a max heap and perform extract max N-1 times.

Heap PUSH is basically insert algo and Heap POP is Extract Min/Max.

INSERT\_MAX\_HEAP

void INSERT\_MAX\_HEAP(vector<int>& stones){

int n=stones.size();

int i=n-1;

int par = (i-1)/2;

while(i>0 and stones[i] > stones[par]){

swap(stones[i] , stones[par]);

i=par;

par =(i-1)/2;

}

}

HEAPIFY MAX HEAP

void heapify(vector<int>& stones , int i){

int n = stones.size();

int l = 2\*i+1;

int r = 2\*i+2;

int largest = i;

if(l<n and stones[l] > stones[largest])

largest=l;

if(r<n and stones[r] > stones[largest])

largest=r;

if(largest!=i){

swap(stones[i],stones[largest]);

heapify(stones,largest);

}

}

Priority Queue STL in C++

priority\_queue<int>pq; //Max heap by default

priority\_queue<int,vector<int>,greater<int>>pq; //Min heap

We can use our own comparator to set priority like sort algorithm;

1. Kth LARGEST ELEMENT ( On+KlogN) nlogn in worst case. Quick select performs better.

int findKthLargest(vector<int>& nums, int k){

int n=nums.size();

priority\_queue<int>pq;

for(int i=0;i<n;i++)

pq.push(nums[i]);

while(k--){

int cur\_max= pq.top();

pq.pop();

if(!k)

return cur\_max;

}

return pq.top();

}

1. Top K frequent element

**Input:** nums = [1,1,1,2,2,3], k = 2

**Output:** [1,2]

Approach – Make a amp with frequency, makemax heap of pair of element and frequency and perform extract min k times;

struct myComp{

bool operator()(pair<int,int>a , pair<int,int>b){

return a.second<b.second;

}

}; // Method to make max heap in order of second element of pair.

vector<int> topKFrequent(vector<int>& nums, int k) {

int n=nums.size();

map<int,int>freq;

for(int i=0;i<n;i++)

freq[nums[i]]++;

priority\_queue<pair<int,int>,vector<pair<int,int>>,myComp>pq;

for(auto x : freq){

pair<int,int> temp = {x.first , x.second};

pq.push(temp);

}

vector<int> ans;

while(k--){

pair<int,int> curr = pq.top();

pq.pop();

ans.push\_back(curr.first);

}

return ans;

}

3 SLIDING WINDOW MAXIMUM

Given an array and a window of size k. return max element in window from left to right.

**Input:** nums = [1,3,-1,-3,5,3,6,7], k = 3

**Output:** [3,3,5,5,6,7]

**Explanation:**

Window position Max

--------------- -----

[1 3 -1] -3 5 3 6 7 **3**

1 [3 -1 -3] 5 3 6 7 **3**

1 3 [-1 -3 5] 3 6 7  **5**

1 3 -1 [-3 5 3] 6 7 **5**

1 3 -1 -3 [5 3 6] 7 **6**

1 3 -1 -3 5 [3 6 7] **7**

Approach 1: Using Max Heap. Create a heap of pair of value,index. If index root is Out Of Bounds, Pop IT. Return max at every step.

vector<int> maxSlidingWindow(vector<int>& nums, int k) {

if(k==1)

return nums;

int n=nums.size();

vector<int>ans;

priority\_queue<pair<int,int>>pq;

//push the first k element in heap.

for(int i=0;i<k;i++){

pq.push({nums[i],i});

}

ans.push\_back(pq.top().first)

for(int i=k;i<n;i++){

// keep poping top till it is Out Of Bounds.

while(!pq.empty() and pq.top().second <= i-k)

pq.pop();

pq.push({nums[i],i});

ans.push\_back(pq.top().first);

}

return ans;

}

Approach 2(Best) : Using Dobly Ended Queue. Maintain it in descending order. Keep poping top till it is Out Of Bounds.

4 K-SORTED ARRAY

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time. For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

***Input :****arr[] = {6, 5, 3, 2, 8, 10, 9}  
            k = 3****Output :****arr[] = {2, 3, 5, 6, 8, 9, 10}*

***Input :****arr[] = {10, 9, 8, 7, 4, 70, 60, 50}  
         k = 4****Output :****arr[] = {4, 7, 8, 9, 10, 50, 60, 70}*

Approach: Create a min Heap of Size K. extract min n times. Remove top until its not out of bounds.

int sortK**(**int arr**[]**, int n, int k**)**

**{**

// Insert first k+1 items in a priority queue (or min heap)

//(A O(k) operation). We assume, k < n.

priority\_queue**<**int, vector**<**int**>**, greater**<**int**>** **>** pq**(**arr, arr + k + 1**)**;

//sorting inplace with no extra memory.

int index = 0;

for **(**int i = k + 1; i **<** n; i++**)** **{**

arr**[**index++**]** = pq.top**()**;

pq.pop**()**;

pq.push**(**arr**[**i**])**;

**}**

while **(**pq.empty**()** == false**)** **{**

arr**[**index++**]** = pq.top**()**;

pq.pop**()**;

**}**

**}**