Given an array we want to do the following operations on array.

→ update (A, i, x) which will set A[i] = x

→ Range minimum Query - RMQ(A, i, j)

which will give $\min_{i \le k \le j} A[k]$ such that

For ex:-

indices of A →	0	1	2	3	4	5	6	7	→ i, j
values →	4	7	3	2	8	12	1	16	

if i = 1 j = 7 then you should return index (k) = 6

∵ it holds the least value (1)

if i = 1 j = 4 they you should return index (k) = 3

∵ it holds the least value (2)

So, now "update" is basically, we can update index 6 ~~to~~ from 1 to 14. If we make it 14 & then we ask the question for RMQ for i = 2 to j = 7 the it should return 3 & NOT 6.

We need a data structure (DS) which will perform these operations efficiently

If we just use an array then the update will happen in $O(1)$, while RMQ will take $O(n)$.

So if we do n operations
where $\frac{n}{2}$ operations are update

& $\frac{n}{2}$ operations are RMQ

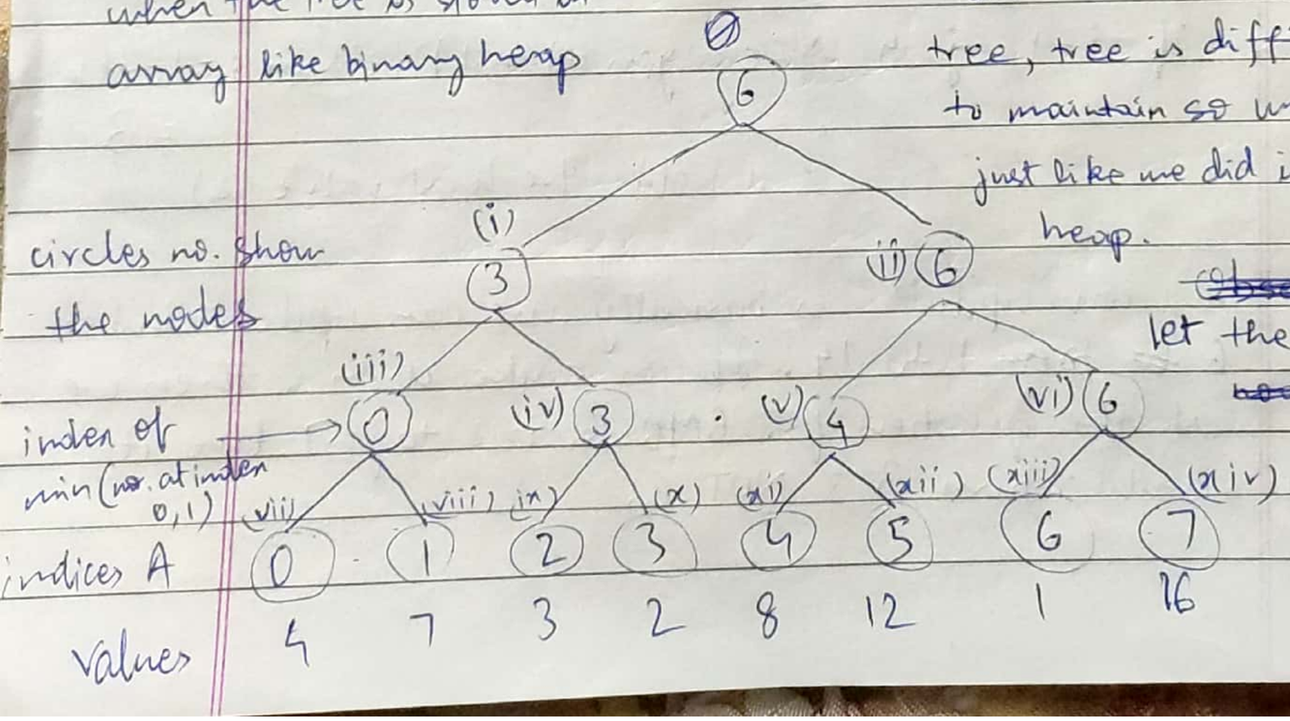
then the order will turn out to be $O(n^2)$

∴ ~~wel~~ we'll study a DS (Segment tree) which will enable us to do both operation in order $\log n$.

SEGMENT TREE

(i), (ii), (iii) --- are indices ~~of~~ when the tree is stored in array like binary heap

So, now we don't want to maintain this tree, tree is difficult to maintain so we do just like we did in binary heap.



~~Observe that~~ let the no. of ~~nodes~~ leaf nodes $= n$

~~But no. of tree~~ Here, n = no. of inputs in array
but if no. of inputs $\neq 2^k$
then this is not true.

Date: / /
Page No.:

Assuming the tree to be complete.

~~We're~~ We want to assume that the array that is given to us is a power of 2, if it is not a power of 2 then we won't get a nice tree like ~~a~~ given in previous pg.

So, if it is not a power of 2 then we add enough dummy variables in the end

some very large no. (which we'll call ∞)
but when we'll implement it we'll just write a very large no.

For ex:- if no. of inputs in array = 11
then we'll make it 16.
where 12, 13, 14, 15 & 16 will have ∞.

$\because$ we will have to add dummy variables
then n > no. of inputs in array.

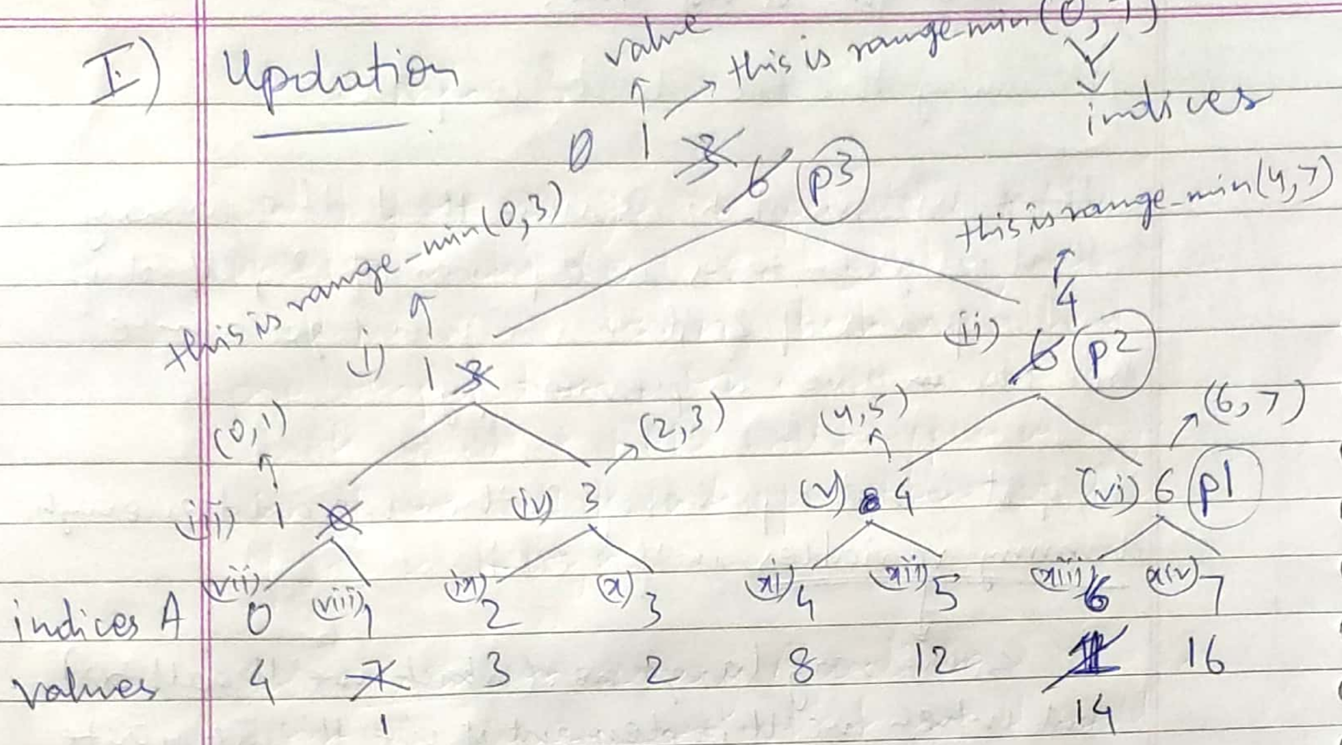
no. of internal nodes = n-1, one less than the no. of leaf nodes.

$\Rightarrow$ total no. of nodes = 2n-1.

Now, lets assume that we want to update the 6th index from ~~8~~ 1 to 14

Date: / /
Page No.:

I) Updation



step 1 we'll change the val at ind 6 to 14

step 2 " " go to its parent (p1) & see which of its child now has lower value.

step 3 Then we'll go to parent of p1 i.e. p2 & check again. We see that now val at ind 4 is smaller

step 4 Again, we'll go to parent of p2 and check. We see that now val at ind 3 is smaller.

Order of this whole operation i.e. updation will be logn (n = no. of leaf nodes)

* lets perform another updation
we'll change val at ind. 1 from 7 to 1

II.) ~~So, if an~~ Range-min-Query

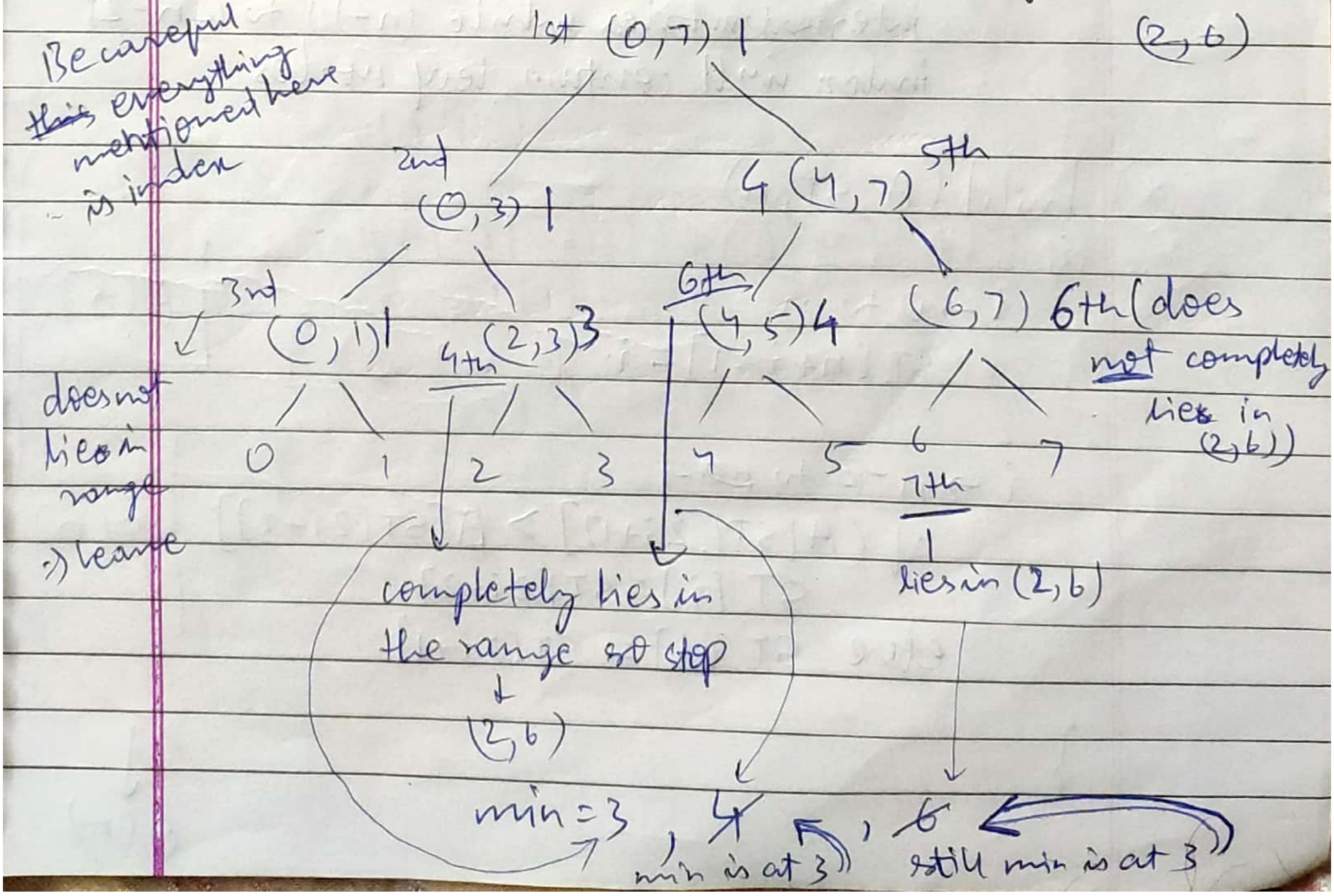
So, if an internal node is from range (s, e) ↓ start ↓ end

$$m = \frac{s+e}{2}$$

then the node's left child will have range (s, m) & its right child will have range (m+1, e)

Lets say we want to find RMQ(2, 6)

Now, each node implicity shows a range as shown below corresp. to the tree on prev. pg.



$\therefore$ min = 3

But before we go to update & RMQ, we need to build the tree. Building the tree will be of order n, which is called precomputation. Once we build the tree both update & RMQ will happen in logn. So, if we call k such operations it'll be of order k logn.

So given an array A will be a tree in form of array, ~~so~~ which we'll call ST (Segment Tree).

we see that there'll be n-1 internal node

⇒ 0 to (n-2) ~~iden~~ index of ST will contain internal nodes while (n-1) to (2n-2) index will contain leaf nodes

(*) build (A, ST, n)

```
build (A, ST, n)
{
    i -> 0 to n
        ST[n+i-1] = i;                                  θ(n)

    i -> n-2 to -1
        if (A[ST[2i+1]] > A[ST[2i+2]]
            ST[i] = ST[2i+2]                            θ(n)
        else ST[i] = ST[2i+1]
}
```

$\theta(n)$

37:24

Date: / /
Page No.:

~~Max~~. If no. of inputs = m
then max. no. of leaf nodes $\approx m$
$\Rightarrow$ ST will acquire space $\approx 4m \approx [2(2m-1)]$
original array space $= m$

$\therefore$ max. space required $\approx 5m$
which is of order m

If you want to locate a value which is at ith index in A in the leaf node in ST then its index will be $(n-1+i)$*.

(*) update (ST, A, i, x)

```
{
    A[i] = x;

    i = (n-1+i-1) / 2

    while(i >= 0)
    {
        if((A[ST[2i+1]]) > A[ST[2i+2]])
            ST[i] = ST[2i+2]
        else
            ST[i] = ST[2i+1]

        i = (i-1) / 2; }
}
```

$O(\log n)$

$\because$ height at max is $\log n$

Date: / /
Page No.:

⊛ RMQI(ST, A, i, j, s, e, p)

```
{
    if (j<s || i>e)      } → handles the
        return n;        }   case when
                             (i,j) does not
                             contain (s,e)

    if (i≤s & e≤j)       } → handles the case
        return ST[p];    }   when (i,j) completely
                             contains (s,e)

    m = (s+e)/2

    l1 = RMQI (ST, A, i, j, s, m, 2p+1)
    l2 = RMQI (ST, A, i, j, m+1, e, 2p+2)

    if (A[l1] < A[l2]) return l1;
    else return l2;
}
```

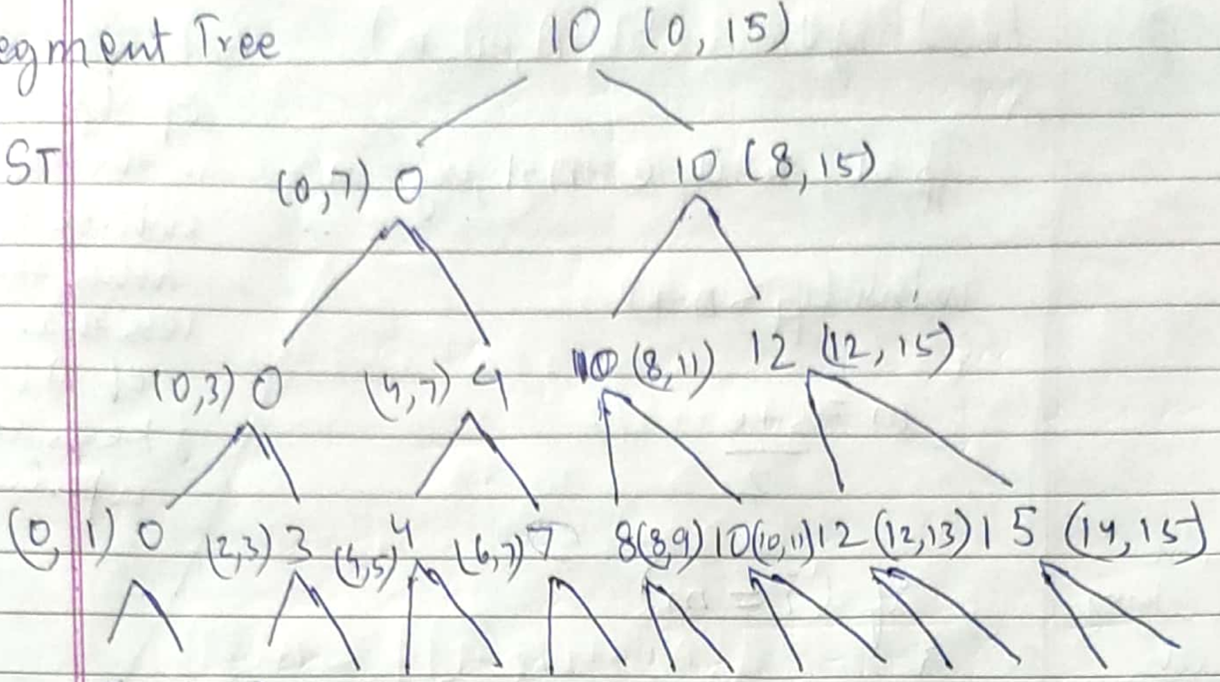
⊛ RMQ(A, i, j, n)

```
{ RMQI(ST, A, i, j, 0, n-1, 0) }
```

Scanned with CamScanner

Segment Tree

ST



indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
values	4	8	9	6	24	32	28	27	30	50	1	4	3	12	18	14

- update (A, i, x)
  $A[i] = x$

- RMQ (A, i, j) $\quad\quad \Theta(\log n)$

  $= \arg\min_{i \le k \le j} A[k]$

- NextRightMin (A, i) → gives the 1st index ~~of~~ on the right of i which holds the no. smaller than A[i].

  ST & n are also its parameters

~~auto~~ $j = i+1$
while($A[j] \geq A[i]$ & $j<n$) j++
return j

n = no. of leaf nodes

Date: / /
Page No.:

(*) NextRightmin (ST, A, i, n)
{

p=0, s=0, e=n-1, j=n;

keeps track of the index in the right subtree which holds the val. less than that of A[i]. j keeps on updating

while (p < n-1)
{

$m = \frac{s+e}{2}$

We're checking each node's children so we don't need to go till the leaf node that's why p < n-1

if (i ≤ m)
{
if (A[ST[2p+2]] < A[i])
j = 2p+2

p = 2p+1; e=m;
}

else { p = 2p+2; s=m+1}
}

$O(\log n)$

while (j < n-1)
{
if (A[ST [2j+1]] < A[i])
j = 2j+1

else
j=2j+2
}

$O(\log n)$

return j-n+1; (∵ j = n-1+i)

—x—

Date: / /
Page No.:

LastMin(A, i) → returns the 1st index of the no. ~~can~~ smaller than A[i] from the end.

```
j = n-1
while(A[j] > A[i])
    j--;
return j
```

Ex1→ from end

indices	0	1	2	3	4	5
values	8	9	7	10	1	12

if i = 1
then the f^n will return 4

Ex: 2

indices	0	1	2	3	4	5
values	8	9	7	10	~~1~~	12

if i = 2 then f^n should return 4

(*) LastMin(A, i)

```
{   p = 0;
    while(p < n-1)
    {
        if (A[ST[2p+2]] ≤ A[i])
            p = 2p+2
        else
            p = 2p+1
    } return p-n+1;
}
```

O(log n)

Similarly, we can find NextLeftMin, Begin-Min

If you want a range-max-query ~~they~~ then you can build a max-seg-tree instead of min. After that ~~you~~ we can ~~ask~~ find the next right-max & we can also find LastMax. ~~Next left min~~

(*) Next Left Min(ST, A, i, n)

```
{
    p=0, s=0, e=n-1, j=n
    while(p<n-1)
    {
        m=(s+e)/2
        if (i>=m)
        {
            if(A[ST[2p+1]] < A[i])
                j=2p+1
            p=2p+2
            s=m+1
        }
        else { p=2p+1; e=m }
    }
    while(j<n-1)
    {
        if(A[ST[2j+2]] < A[i])
            j=2j+2
        else
            j=2j+1
    }
    return j-n+1
}
```

Date: / /
Page No.:

```
MinfromBegin (ST, A, i, n)
{
    p = 0
    while (p < n-1)
    {
        if (A[ST[2p+1]] <= A[i])
            p = 2p+1
        else
            p = 2p+2
    }
    return p-n+1
}
```

—x—

Rangesumquery → RSQ (A, i, j)

↓

returns the sum of no. from i to j.

So, we do this in the following way

→ The leaf nodes will store all the no. & in any internal node we note down the sum of the left subtree & right subtree & construct the tree in linear time.

RSQ

* build_seg_tree(A, ST, n)

```
{
    for (i -> 0 to n)
        ST[n-1+i] = A[i];

    for (i -> n-2 to -1)
        ST[i] = ST[2i+1] + ST[2i+2];
}
```

* update(ST, A, i, x, n)

```
{
    A[i] = x
    ST[n-1+i] = x
    i = ((n-1+i)-1) / 2

    while(i >= 0)
    {
        ST[i] = ST[2i+1] + ST[2i+2]
        i = (i-1)/2
    }
}
```

* RSQ(ST, A, i, j, n)

```
{
    RSQ1(ST, A, i, j, 0, n-1, 0, n);
}
```

(*) RSQ1(ST, A, i, j, s, e, p, n)

```
{
  if (j < s || i > e)
      return A[n]
  if (i ≤ s & e ≤ j)
      return ST[p]

  m = (s+e)/2

  l1 = RSQ1(ST, A, i, j, s, m, 2p+1, n)
  l2 = RSQ2(ST, A, i, j, m+1, e, 2p+2, n)

  return (l1+l2)
}
```

—x—

If we want to ~~implement~~ ans RSQ, we don't need to implement a segment tree, there is an even simpler data structure called binary index tree (BIT).

BIT

There are 2 ~~opre~~ operations that we're going to do:-

i) RSQ.(A, i, j) $\longrightarrow \sum_{k=i}^{j} A[k]$

ii) update (A, i, x) $\rightarrow A[i] = x$

—x—

So to know RSQ we'll not try to find RSQ, we'll try to find the prefix sum. (PS(i))