Array →

0	1	2	3	4	5
-1	3	4	0	2	1.

(1)

Questn → To find min. element-

index 2, 4 → 0.

index 0, 3 → -1.

Alternative Approach → 3×3 matrix.

3	-1	2
~~0~~	~~1~~	~~2~~

	0	1	2
0			
1	3	-1	-1
2	-1.	-1	2.

time takes → $O(n^2)$ to build matrix.

Segment Tree. →

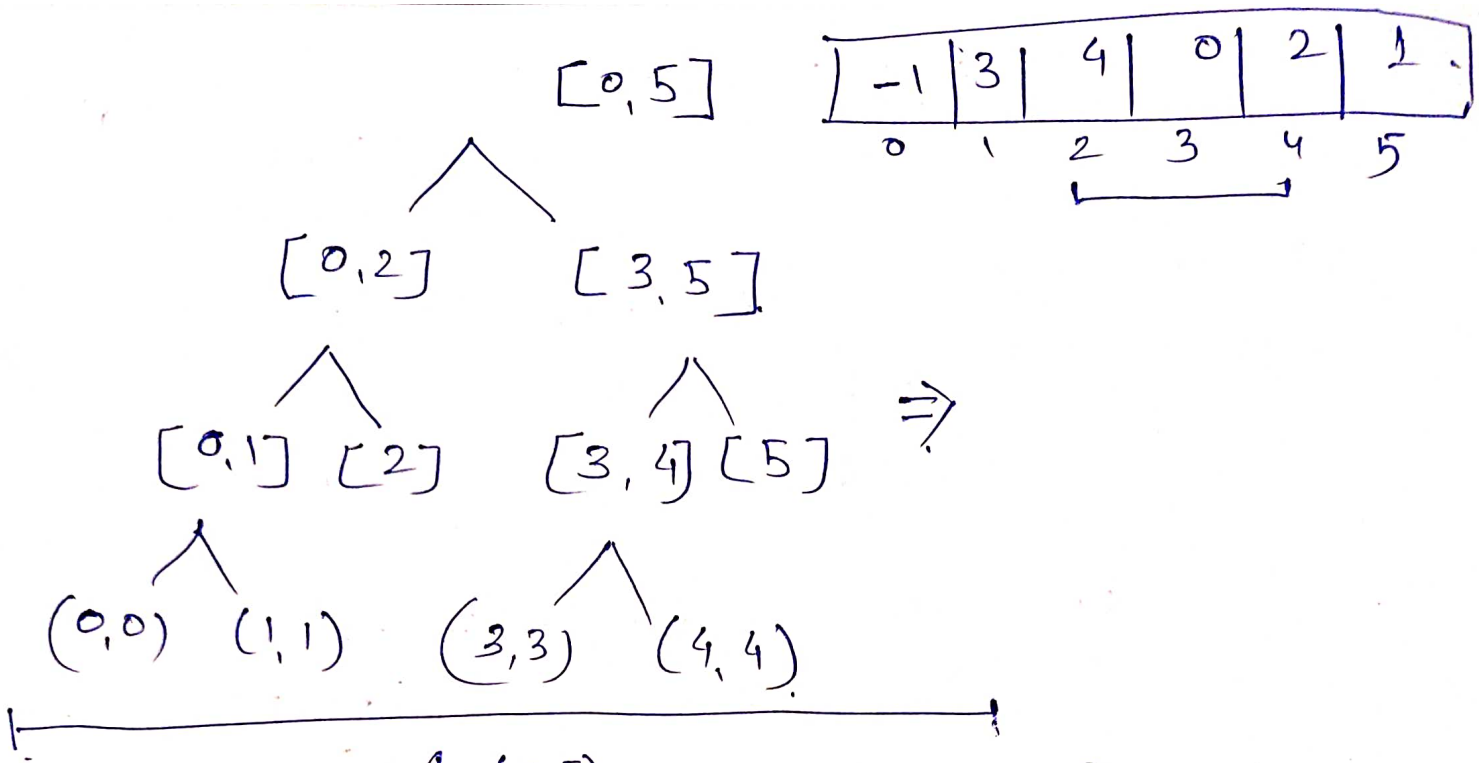
-1	3	4	0	2	1
0	1	2	3	4	5

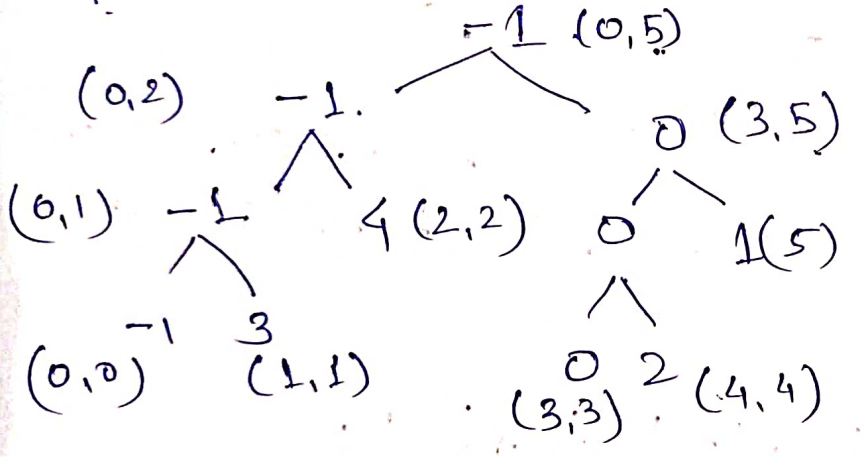
To build. Divide & Conquer Approach.

-1 3 4 / 0 2 1.

-1 3 / 4 / 0 2 / 1.

-1 / 3 4 / 0 / 2 / 1.





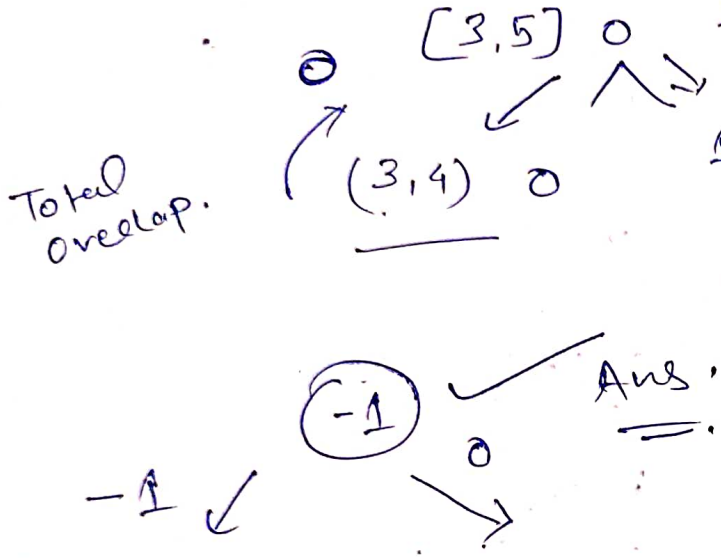
① Partial overlap
② Total overlap
③ No overlap.

Query min. of [0, 4].

① Partial overlap - Go inside.
② Total overlap - return the min. value index.
③ No overlap - Gives the max. value possible

Steps. ① [0,2] Totally falls under (0,4). ②
it returns -1.

② (3,5) Partially overlapp→ search both ways.



---

① Build segment Tree.

segment length

→ Array length (Power of 2) → 2^n ~~-1~~ $\cdot\ 2n-1$

→ Array length (Not Power of 2) → 2^{n+1} ~~-1~~.

$2 \times (n+1) - 1$.

2×2.

Ex. * length of Array - 4

size of seg. tree → $2^{1}-1 = 7$.

* length of array - 5

size of seg. tree → $2 \times (2^3) - 1$

$= 8 \times 2 - 1$

$= 15$.

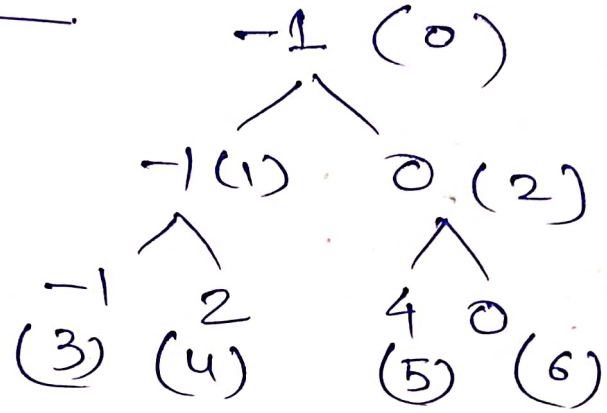
Array

-1	2	4	0
0	1	2	3

Segment Tree.
Array.

+	+	Max	Max	Max	Max	Max.	
0	1	2	3	4	5	6	7

Conceptual one

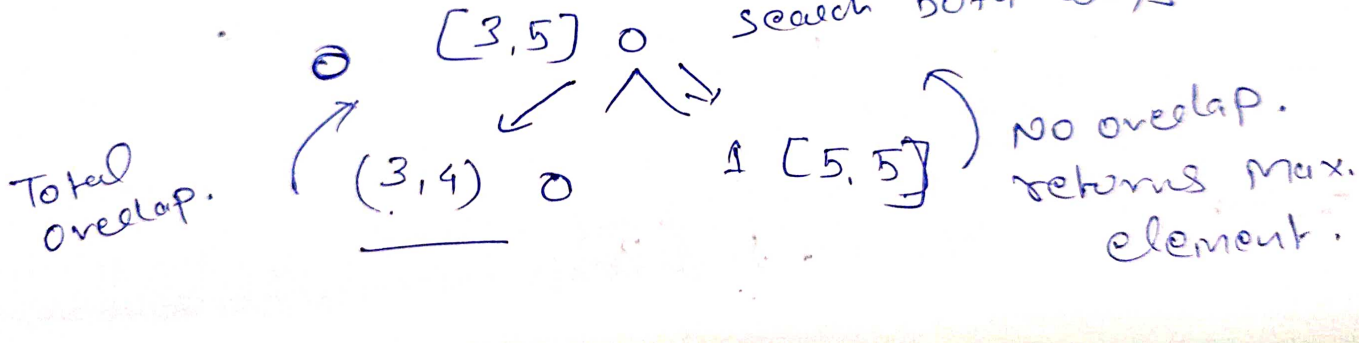


time complexity - To build - O(n)

- To search - O(log n).

Steps. ① [0,2] Totally falls under (0,4). ② it returns -1.

② (3,5) Partially overlapp→



```
void buildTree (int input[], int seg.[], int low, int high, int Pos)
    if (low == high)
{       seg [Pos] = input [low] return; }

    int mid = (low + high)/2 ;
1.  build tree (input, seg, low, mid, 2*pos+1)   → left child
2.  build tree (input, seg., mid+1, high, 2*Pos+2)   → right child.
    seg.Tree[Pos] = min.[seg tree [2*pos+1], seg tree [2*Pos+2]]
}
```

-1	2	4	0	Array
0	1	2	3	

low	high	Pos	Mid	Line No.
0	3	0	1	1.
0	1	1.	0	1. 2.
0	0	3.		return.
~~0~~	~~*~~	~~*~~		return.
1	1	4		

seg. Tree.

	-1	-1	2			
0	1	2	3	4	5	6.

③ ① ②

↑ Pos.

(root of segment tree)

1

6

low	high	pos	mid	line NO.	
0	3	0	1	2.	(second half of seg. tree)
2	3	2	2.	1	
2	2	5	:	return	
2	3	2	2	2	
3	3	6.	return.		
2	3	2	2	2.	

seg. tree.

-1	-1	0	-1	2	4	0
0	1	2	3	4	5	6.

Array

-1	2	4	0
0	1	2	3

Min. [4, 0] → 0.

→ time - O(n).

→ size of seg. Tree - O(n).

```
int range query (int seg tree[], int qlow, int qhigh, int low, int high
                 , int pos )

if (qlow < low && qhigh ≥ high) Total overlap.
    return seg. tree [pos];                    query - [1 - 3]
                                               seg.t. [2 3]
else if (qlow > high || qhigh < low.) // No overlap.
    return max. value ;

int mid = (low + high) /2;
return min. {( rang query (seg tree, qlow, qhigh, low, mid, 2*pos+1)
               rang query (seg tree, qlow, qhigh, mid+1, high, 2*pos+2)

    ?.
```

Sum of Given Range.

① Range Sum Query →

0	1	2	3	4
1	2	3	4	5

Sum (2, 4) = 12.

Sum (1, 3) = 9.

Sum (0, 4) = 15.

Approach

(A) Brute force.

Time = O (N * Q)

Q no. of Queries. ↑ for Each Query.

(B) Preconstruct the Array.

0	1	2	3	4
1	3	6	10	15

Sum (1, 3) = Sum [3] - Sum [0]

= 10 - 1 = 9.

Sum (L, R) = Sum [R] - Sum [L-1].

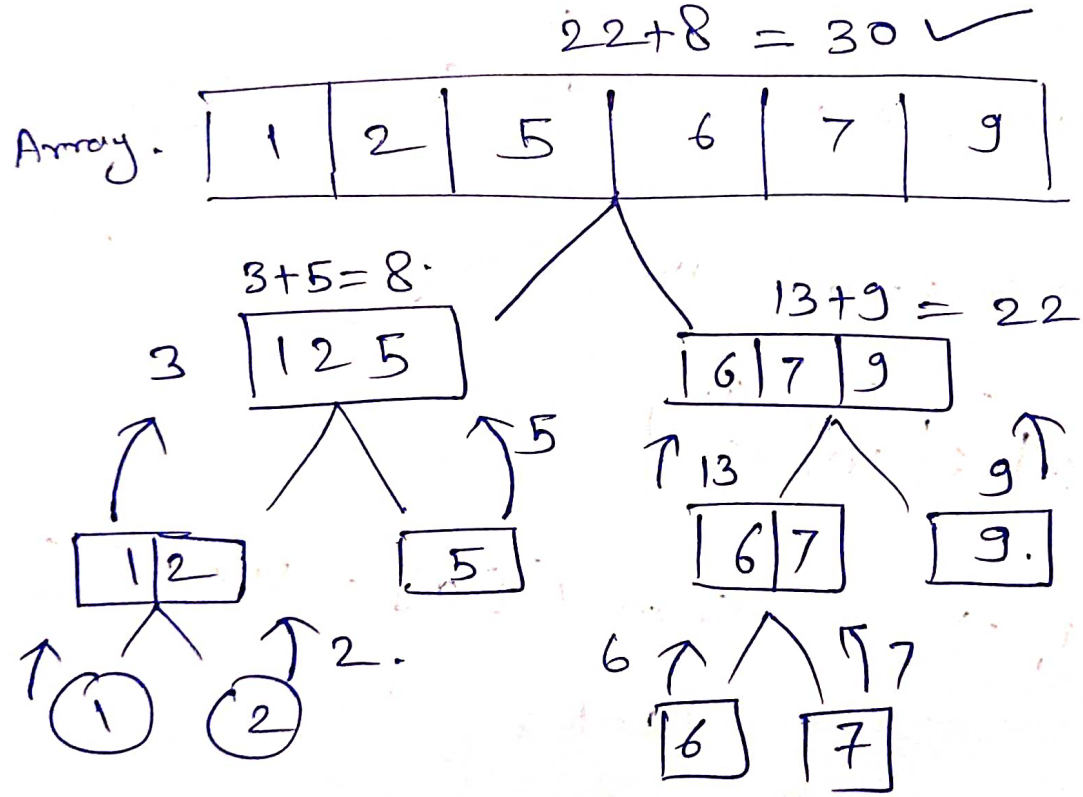
→ Each Query will take O(1) time.

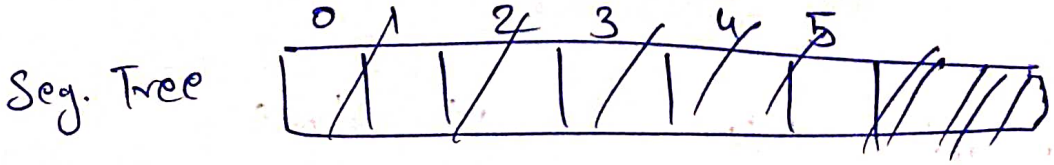
→ for Q. Queries → O(Q) times.

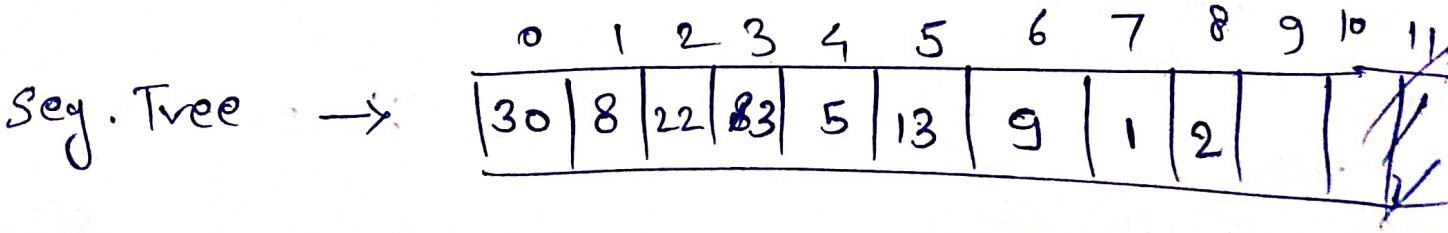
		update.
Brute force.	O(N.Q)	O(1)
Efficient one.	O(Q.)	O(N)

Not good for update.

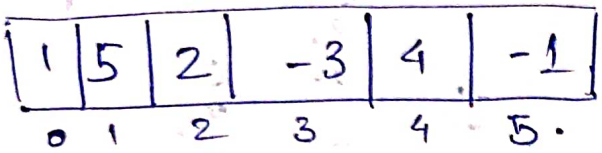
→ for frequent updates we use segment Tree.



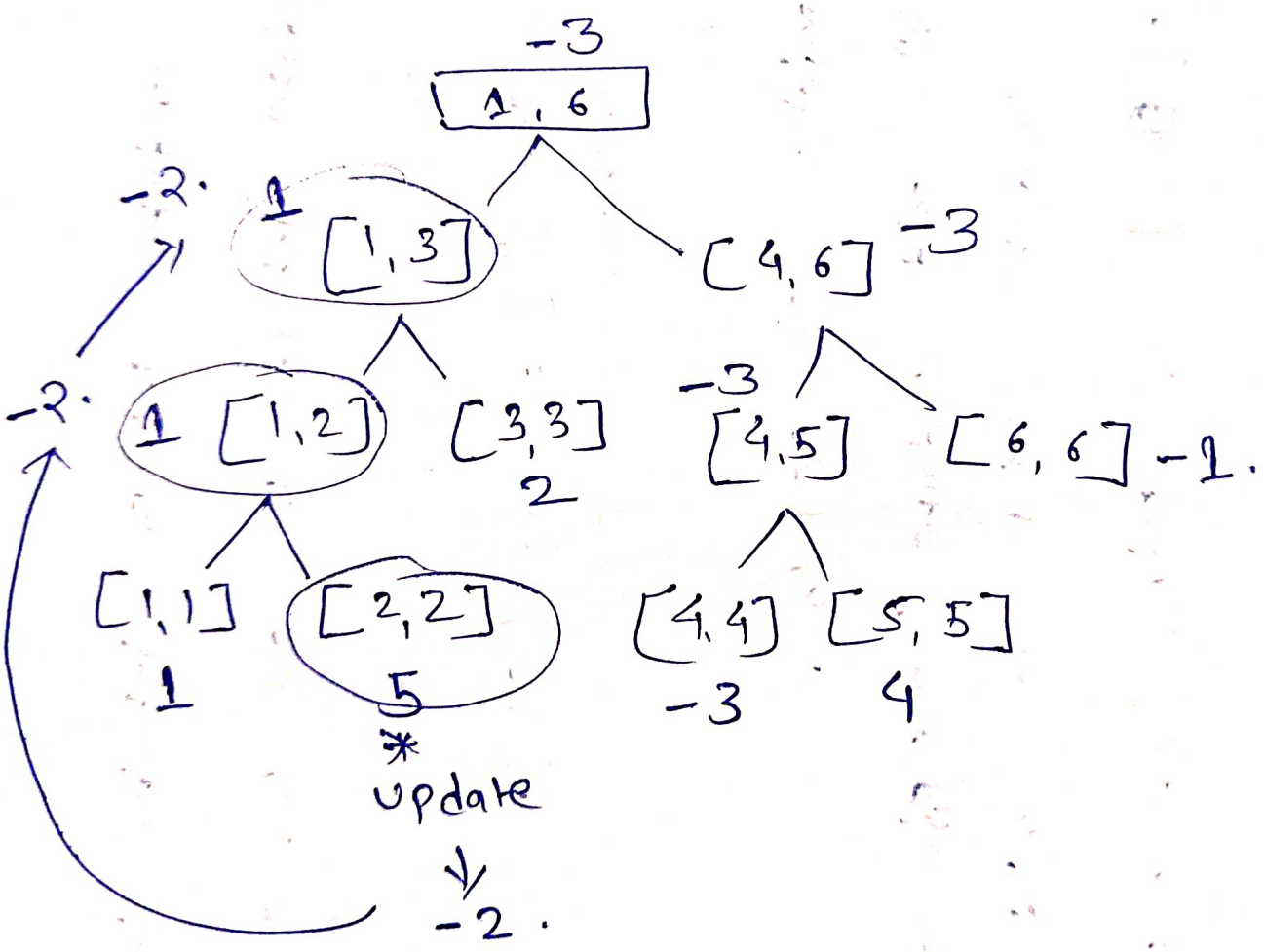




Point update Segment Tree.



Array. update 5 → -2



⇒ 3 Nodes will be affected.

⇒ update - logn time.

query → Point update.

⟹

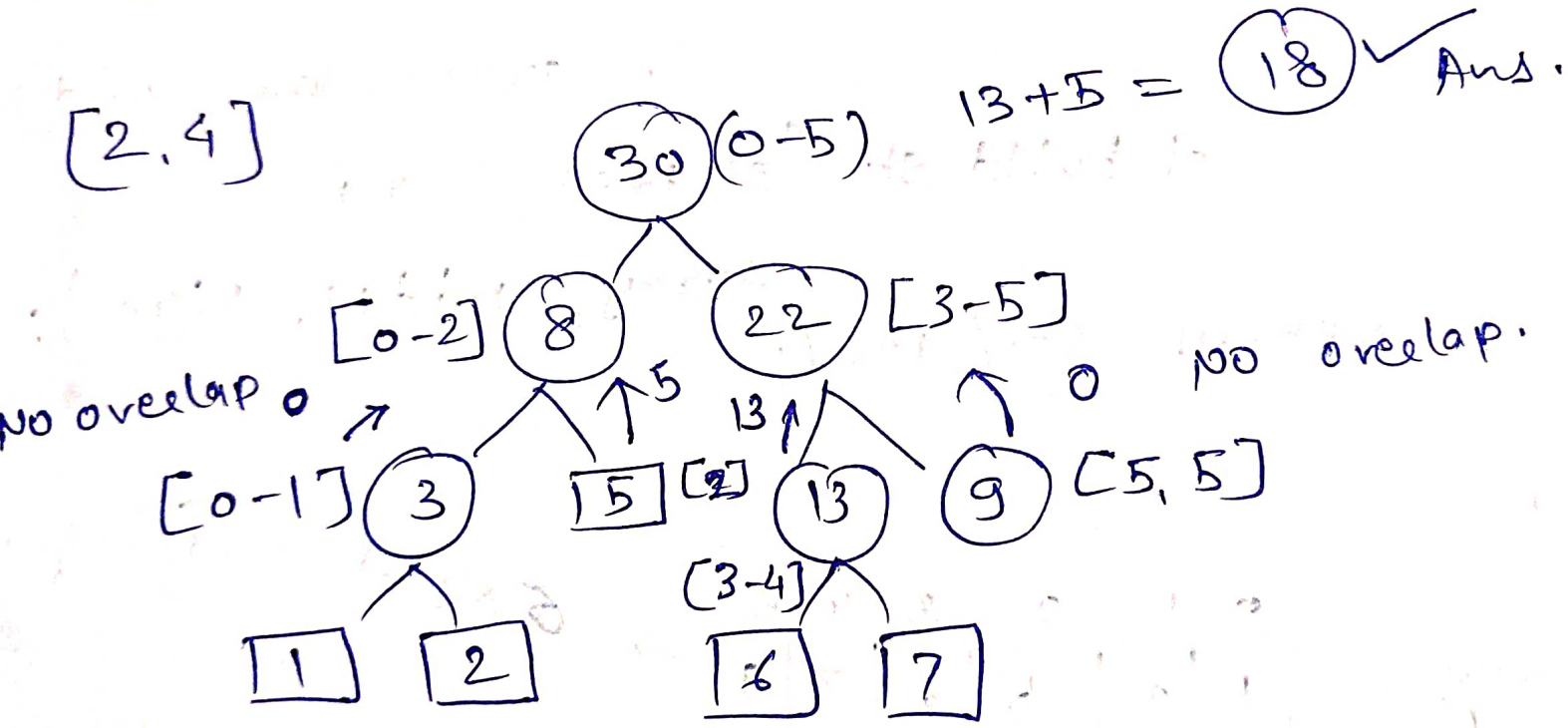
~~void update (int pos, int low, int high~~

```
void update (int pos, int ss, int se, int qi)
{
  if (ss == se)
  {
    st[si] = arr[ss];
    return;
  }
int mid = (ss+se)/2;
if (qi <= mid  update (2*pos, ss, mid, qi);
else  update (2*pos+1, mid+1, se, qi);
  st[pos] = min.(st[2*pos], st[2*pos+1];
}
```

find Range Sum.

find (2, 4). – sum in range [2, 4]

1. Total overlap
2. No overlap
3. Partial overlap.

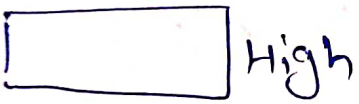


Query Range → l, r.

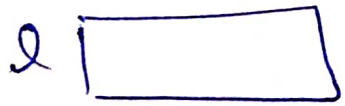
segment range → low, high.

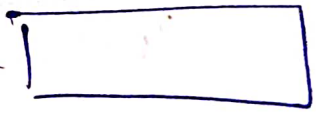
Total overlap.

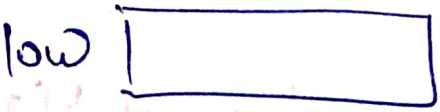
Query Range →



Partial overlap.



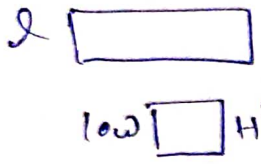




int getsum (pos, low, high, l, r)

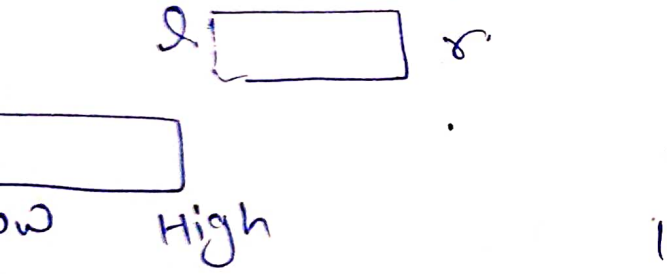
{ if (l ≤ low && r >> high)

return st[pos] Total overlap



if (low < l || high > r)

return 0; No overlap.



else mid (low + High)/2; // Partial.

return getsum (2* pos+1, low mid, l, r)

+ getsum (2* pos+2, mid+1, high
, low high)
l. r.

Range Query.

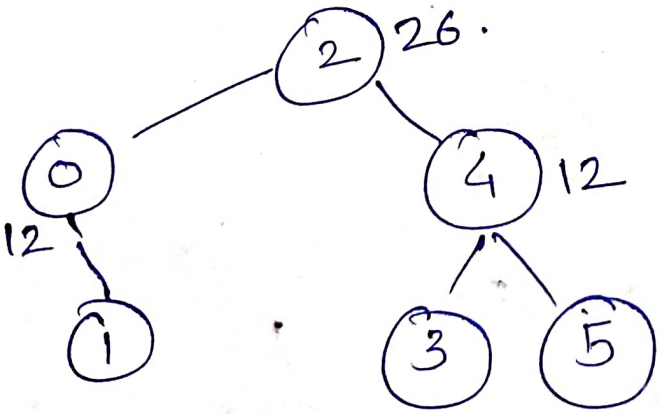
0	1	2	3	4	5	6
4	8	2	5	1		8

1. Range Max. Query - [0, 4] = 1 at location
2. Range Min. Query - [0, 3] = 2 at location.
3. Application of balanced binary Search Tree.

→ time Comx. - O(log n)

0	1	2	3	4	5
4	8	2	5	1	8

Max:



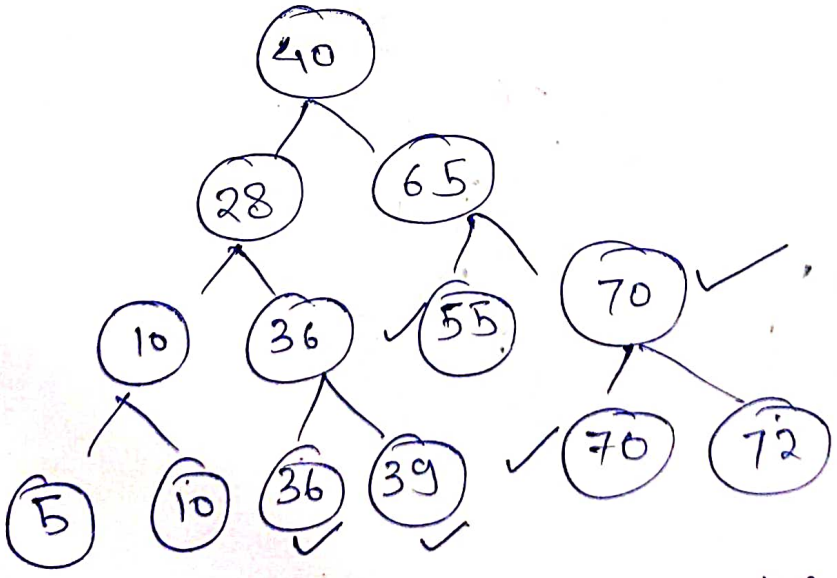
KD Tree.

(1) K dimension Extension of BST.

(2) Process a set of given data points efficiently such that given a range window set of points inside the range can be reported quickly.

(3) More we preprocess & store, faster we can solve query.

(1) 1D. Balanced binary search Tree.



Query [36, 70] → visited nodes, whole subtree is output. —

[36 39 55 70]

Time comx. → depends on visited nodes

KD Tree. (2)

(More than one dimension.)

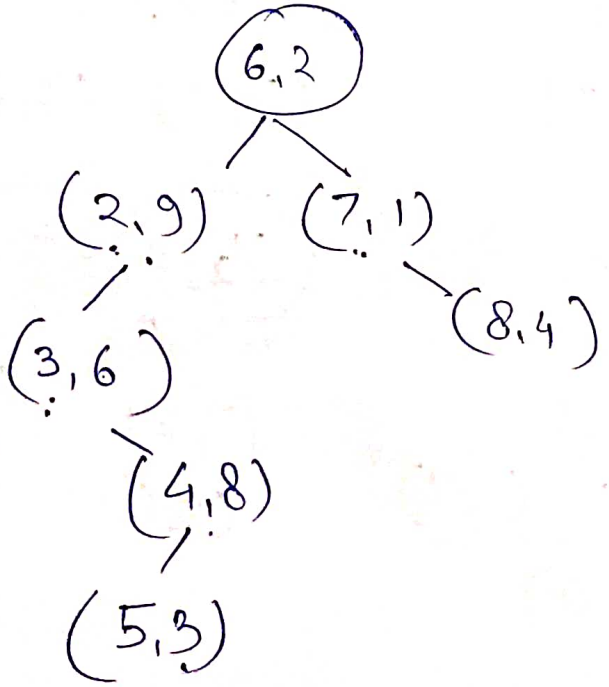
→ split the points alternatively by x planes and y planes.

→ split by x co-ordinate – split by vertical line that has half the points left & right

→ split by y coordinate – split by horizontal line that has half the points ~~left~~ & ~~right~~ below & above.

(6,2) (7,1) (2,9) (3,6) (4,8) (8,4)

(5,3)



x split

y split

x split

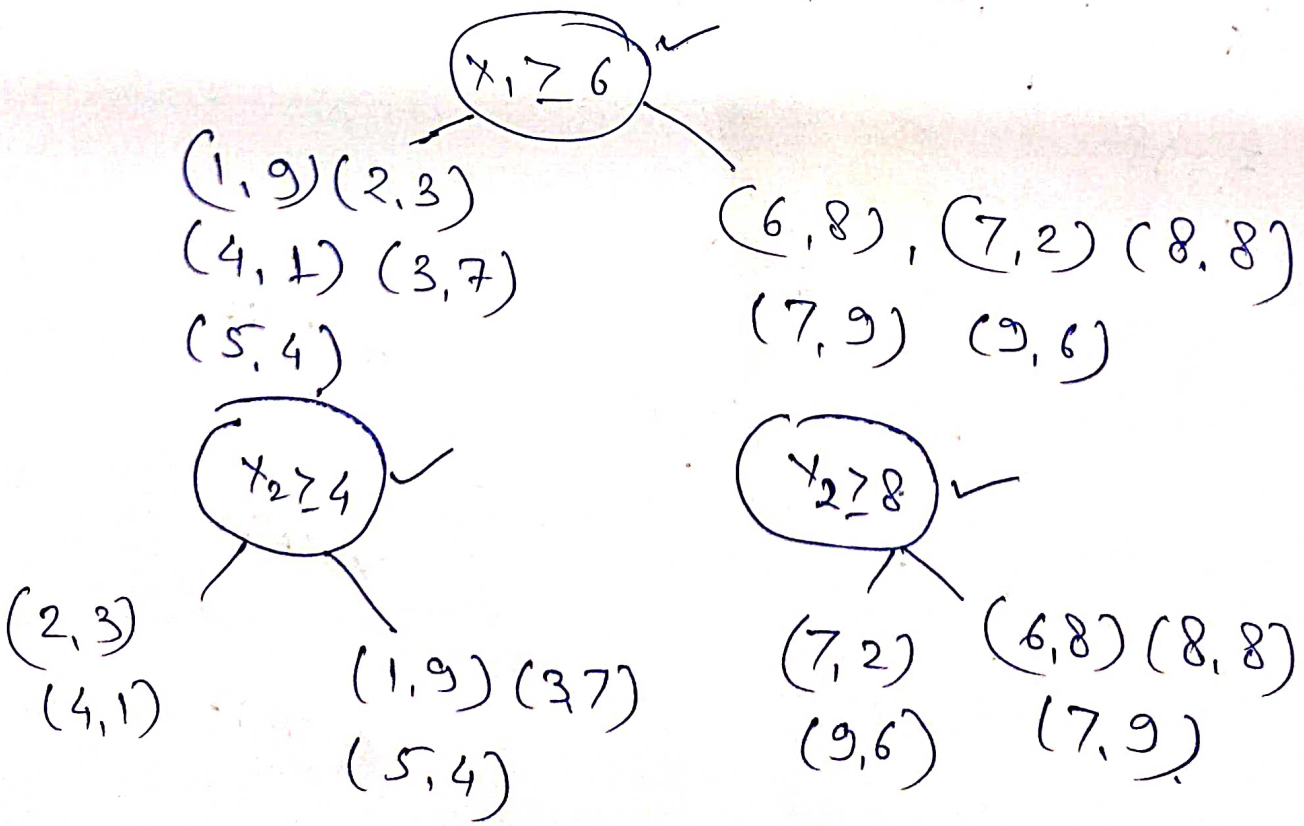
y split.

KD Tree

① Build KD Tree from training data.

- Pick random dimension.
- find median.
- split data.
- repeat..

Ex. (1, 9) (2, 3) (4, 1) (3, 7) (5, 4) (6, 8) (7, 2) (8, 8) (7, 9) (9, 6)

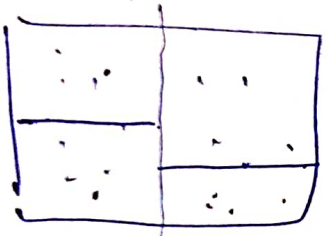


Example.

→ database of millions of people - divided on Height, wt, disease. in hospital.

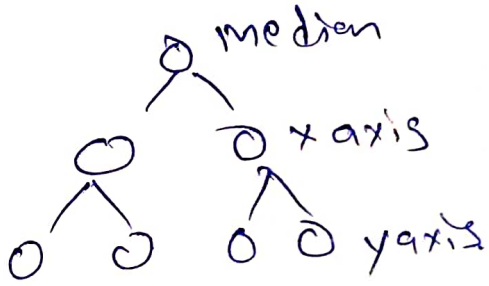
- Search on db of people (70-80) kg. 80cm with heart problem. (

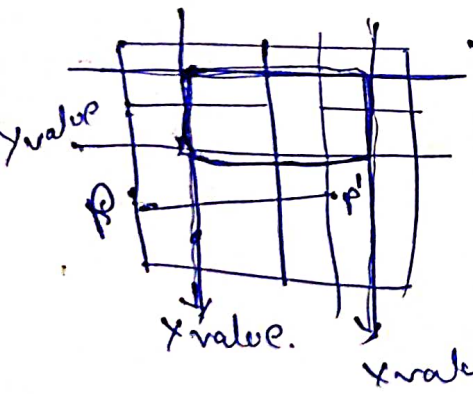
-



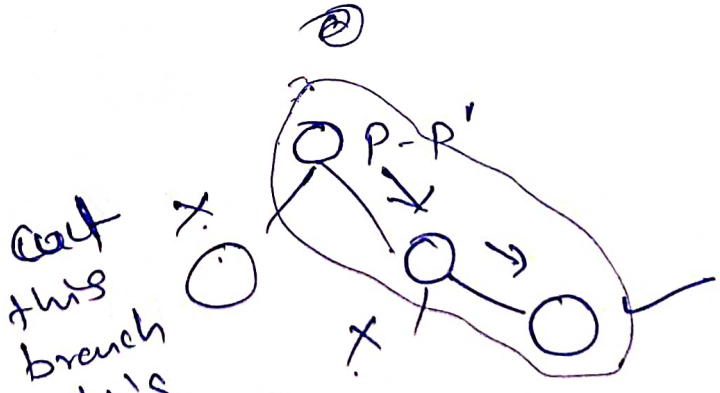
Median along y axis.

with points



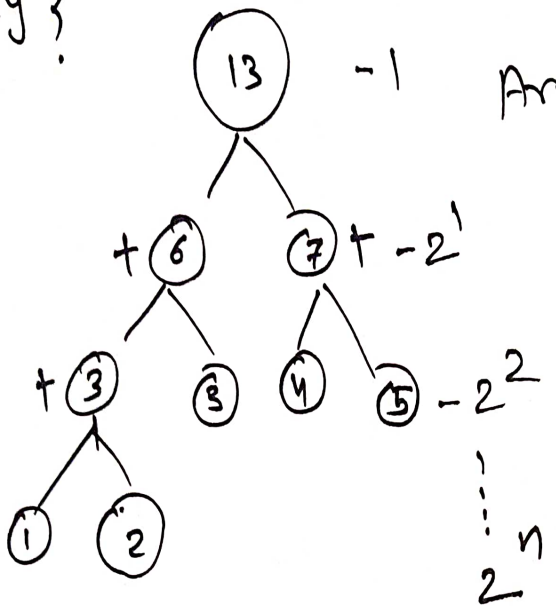


find points b/w (x, y)



(1) Size of segment Array for Array size n $= 4n$.

Why?



Array

1	2	3	4	5
0	1	2	3	4

No. of nodes in segment

$$= 1 + 2^1 + 2^2 + \cdots 2^n = \frac{(2^{n+1} - 1)}{2-1} \leq 4n$$

Safer side.

(1) How to select size →

* when n is Power of 2 → $2n-1$
* when not - $2(\text{next Power of 2}) - 1$

Ex. $n = 6$

seg. tree size $= 2(8) - 1$

$= 15$.

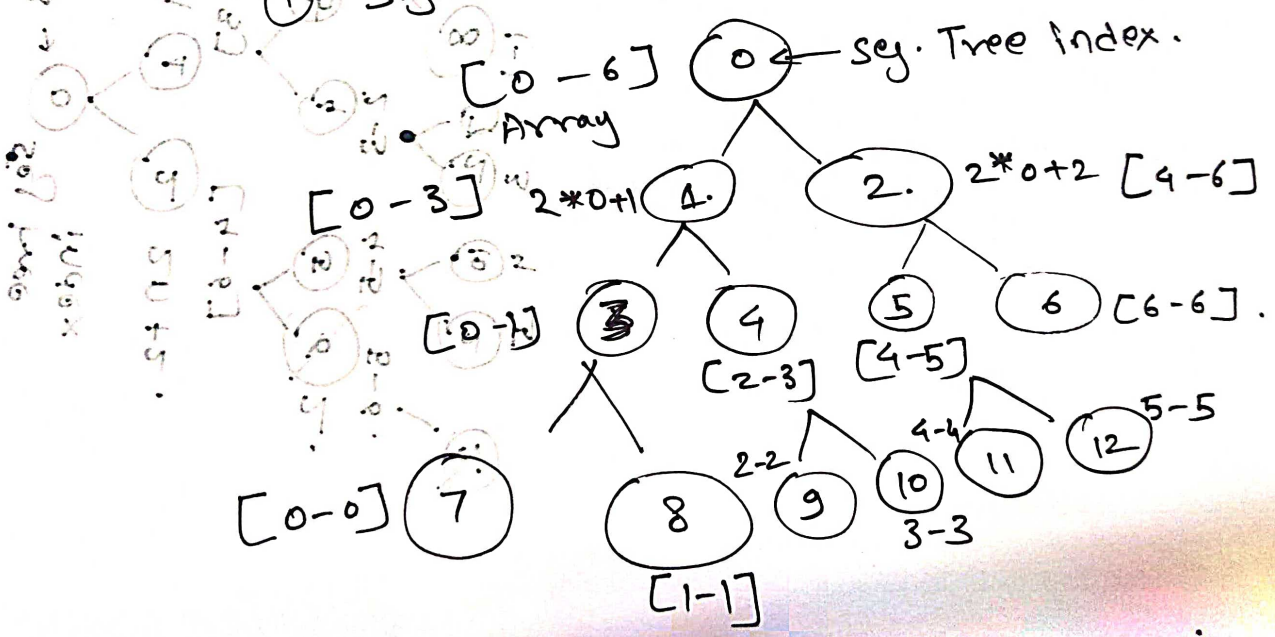
Array —

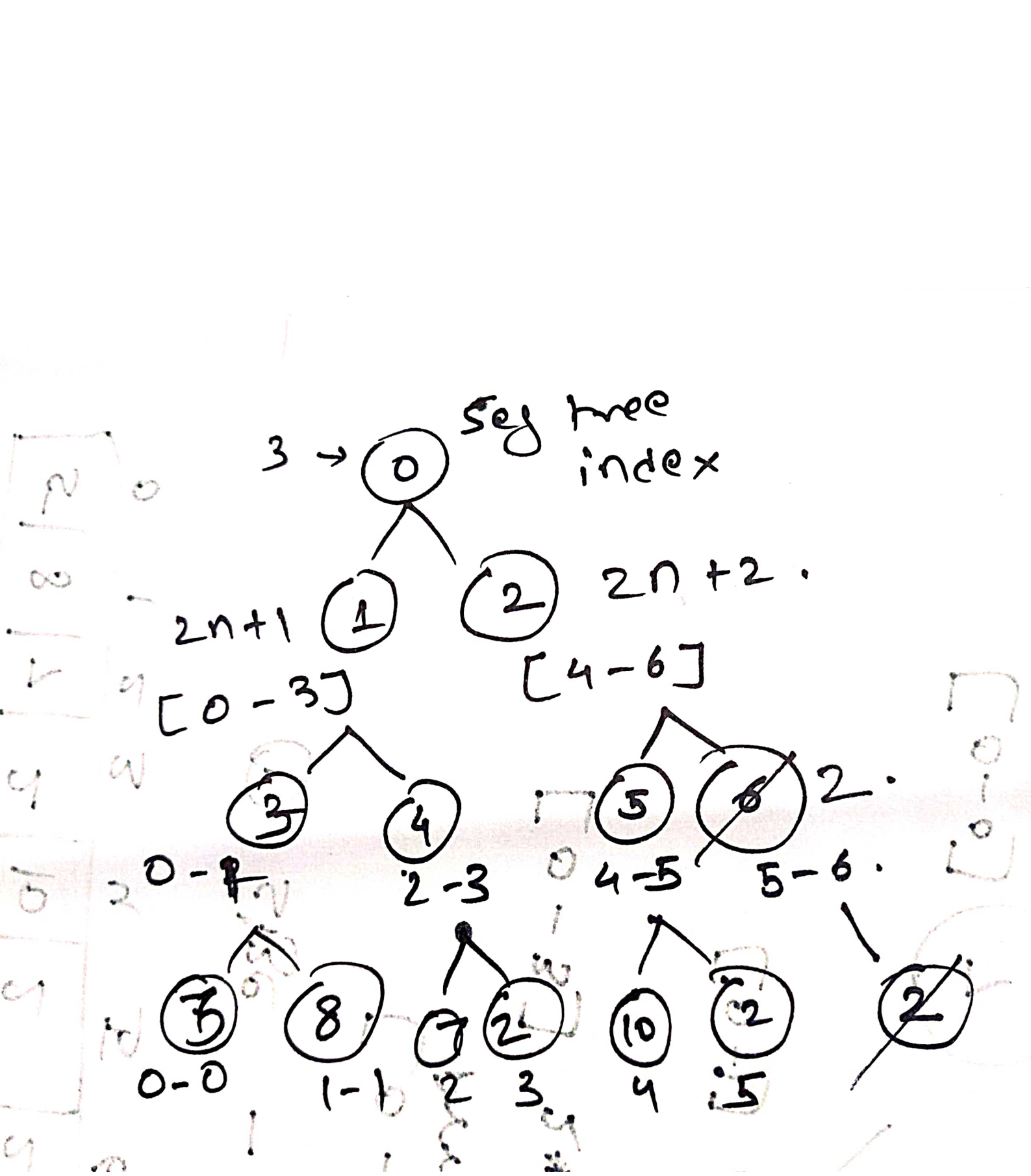
5	8	7	2	10	2	2
0	1	2	3	4	5	6

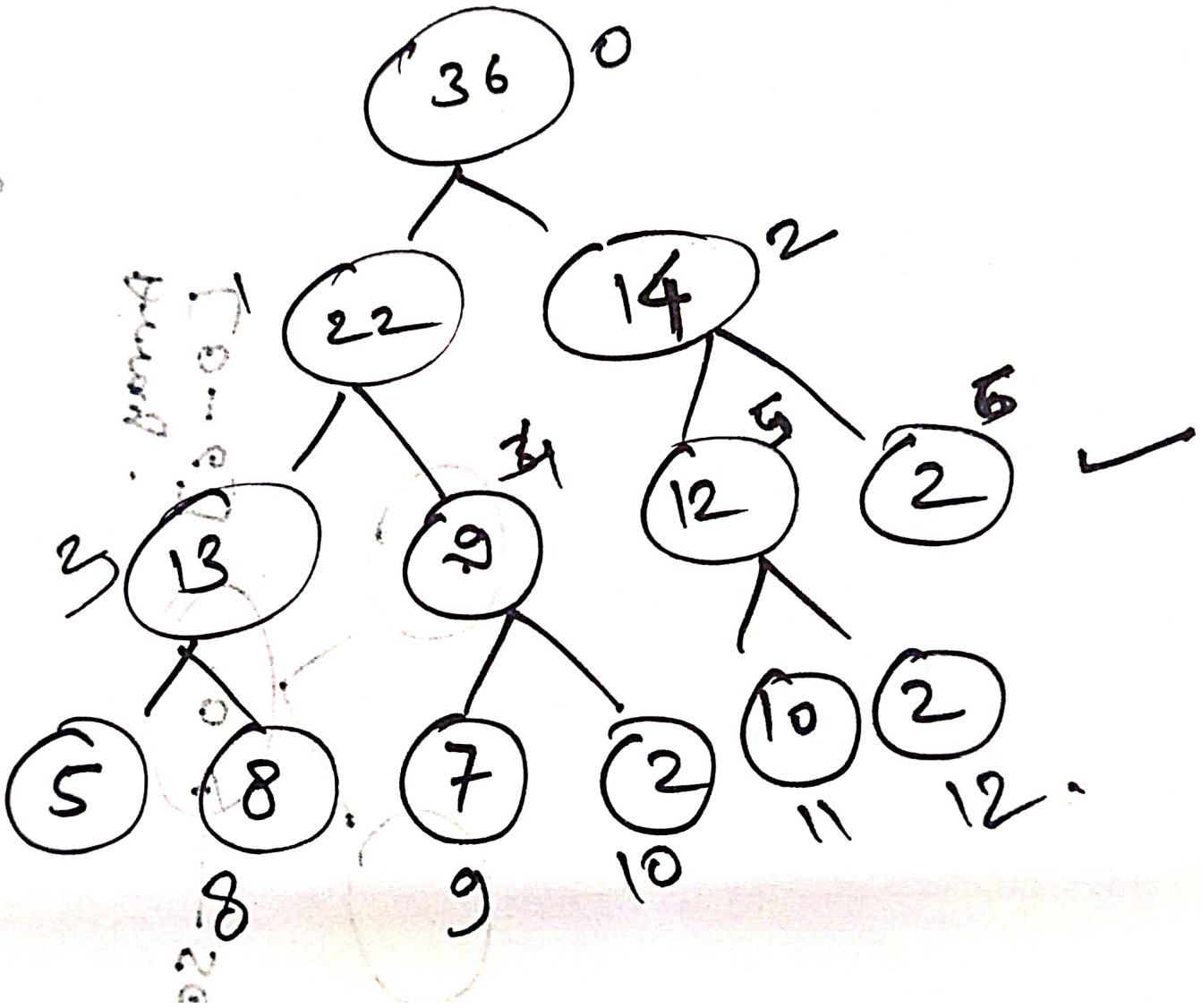
Seg. Tree.

0	1	2	3	4	5	6	7	8	9	10	11	12
36	22	14	13	9	12	2	5	8	7	2	10	2

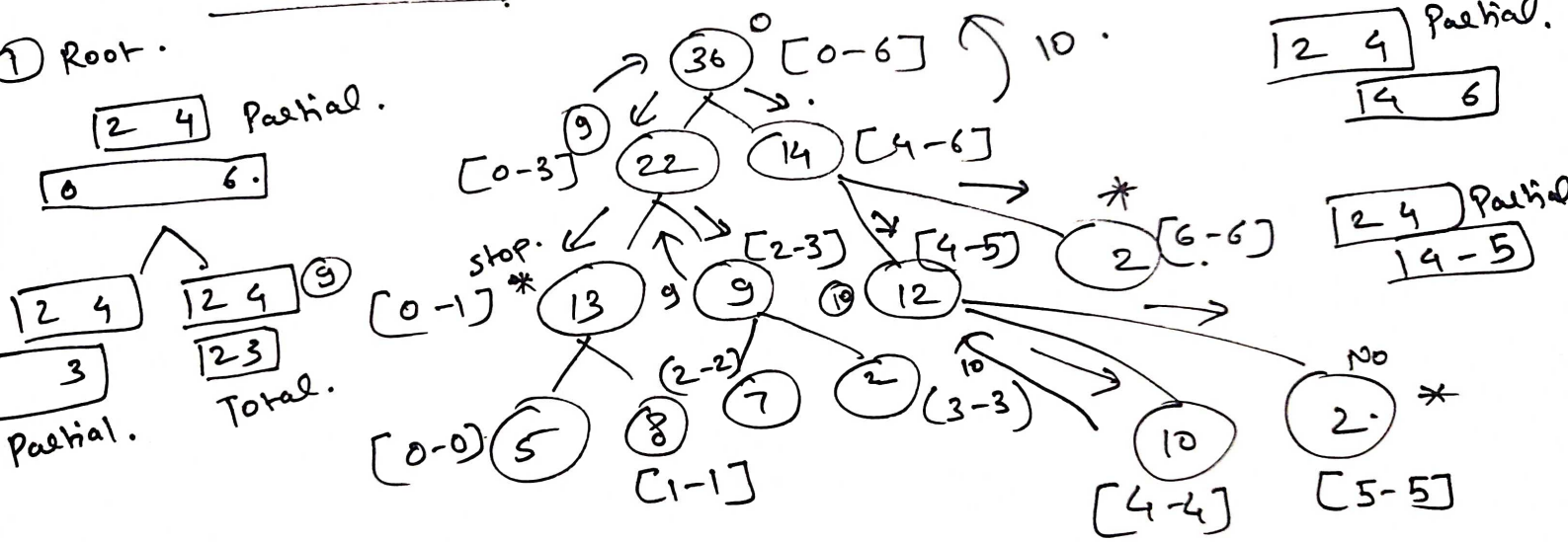
Size $= 7 \times 2 - 1 = 13$.




seg tree
index
3 →
0
1
2
2n+1
2n+2.
[0-3]
[4-6]
3
4
5
6
2.
0-1
2-3
4-5
5-6.
7
8
9
10
11
2
0-0
1-1
2
3
4
5


36
22
14
13
9
12
2
5
8
7
2
10
2

Range-Query sum [2-4].



2 4
01 No overlap.

19 Ans:

9 [↑ 10