Lazy Propagation

Question : Given an array

Queries

1) Range Min [L, R] => Find minimum element in this range

2) Range Update ([L, R], x) => Increment all elements in [L, R] by x

(ind, val) [2, 5], x

	0	1	2	3	4	5	6	7
A =	2	8	7	0	5	1	-2	0

Range Min (2, 5) = -4.

Update Range (1, 3, +4) =>

Range Min (2, 5) = 0

Approach1 :

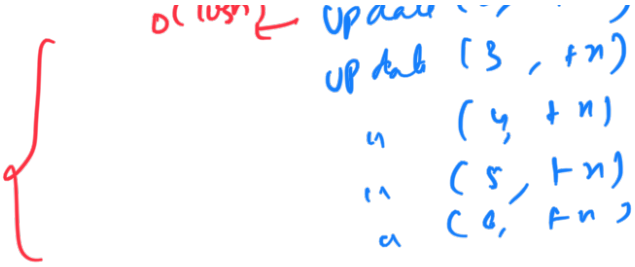
Range Min => O(logn)

Range Update => O(logn) ?

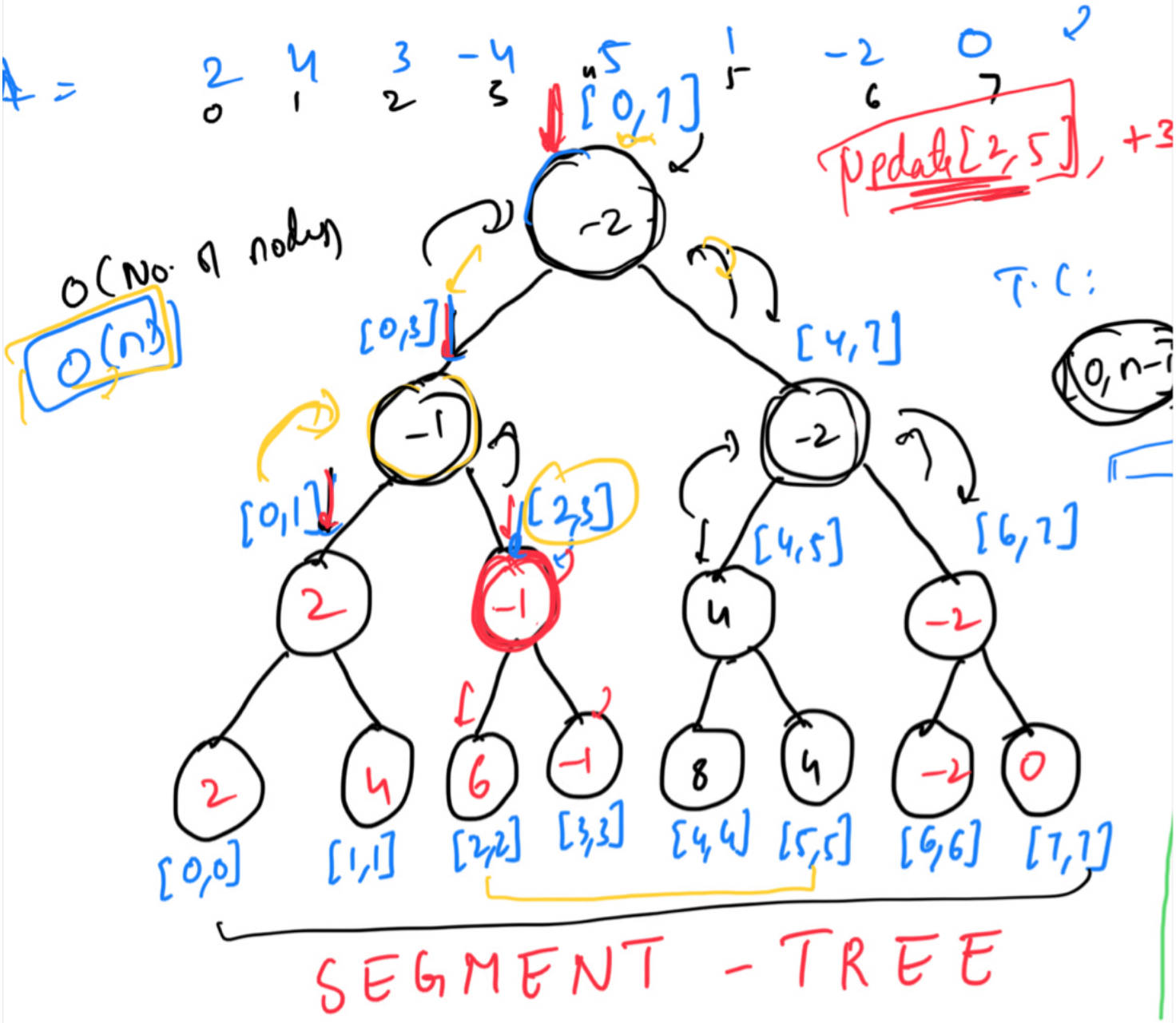
Update (ind, val) => O(logn) [Point update]

[L, R] [2, 6] [0, n-1]

T.C = O(nlogn)



Approach 2:


A = 2 4 3 -4 5 1 -2 0
0 1 2 3 4 5 6 7
[0,7]
update[2,5], +3
O(No. of nodes)
O(n)
T.C:
0,n-1
-2
[0,3]
[4,7]
-1
-2
[0,1]
[2,3]
[4,5]
[6,7]
2
-1
4
-2
2
4
6
-1
8
4
-2
0
[0,0]
[1,1]
[2,2]
[3,3]
[4,4]
[5,5]
[6,6]
[7,7]
SEGMENT - TREE

Approach : Lazy Propagation

→ If there is a complete overlap, Just tell the children of this about the update and return.

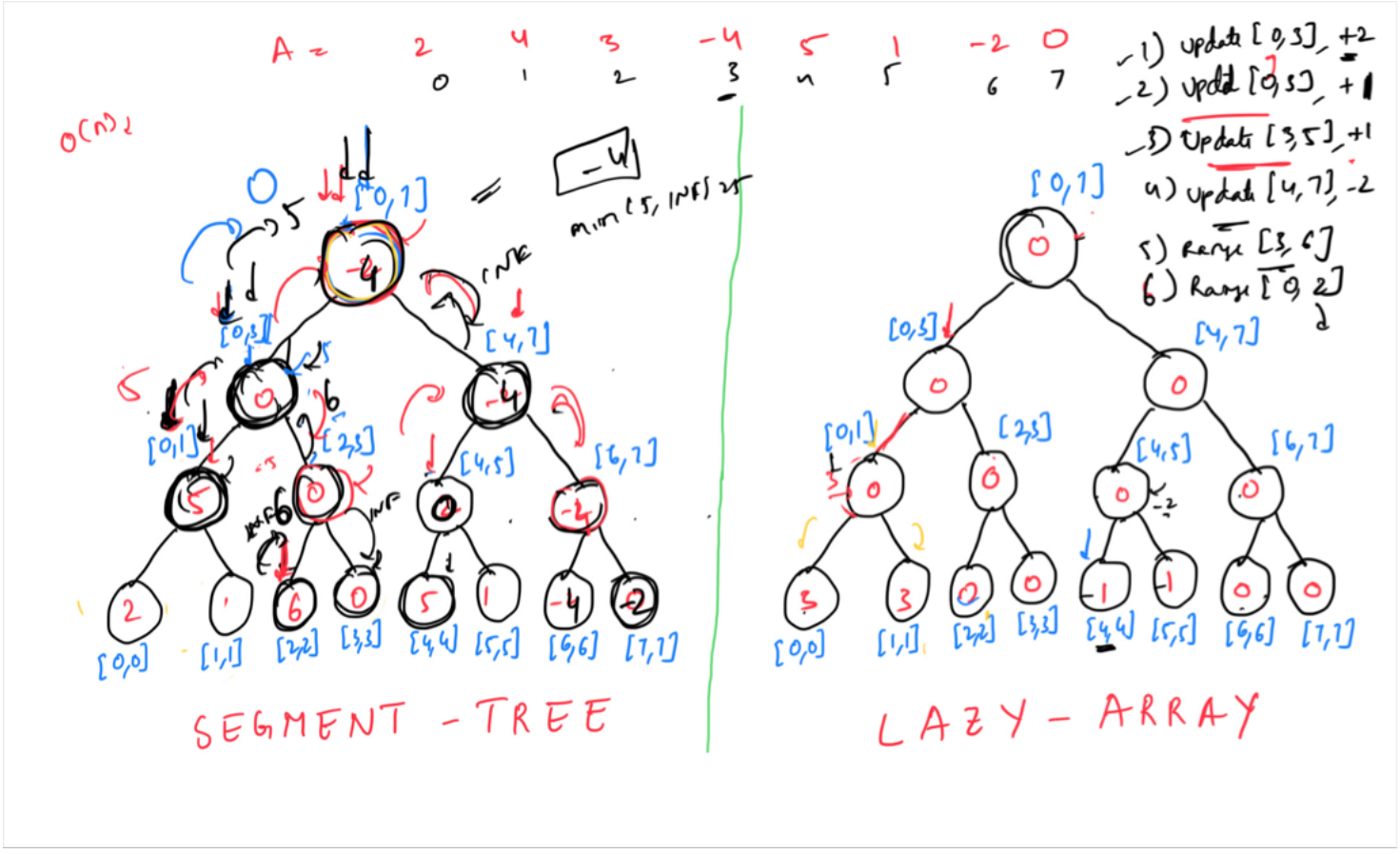
Steps

1) Take another array lazy[] which is of the same size as segment tree.

lazy[4N] = {0}
seg[4N]

2) Initially all the values are initialized to zero

3) In case of complete overlap, we update the value of that node and tell its children about the update through the lazy array


A = 2 4 3 -4 5 1 -2 0
0 1 2 3 4 5 6 7
1) Update [0,3] +2
2) Update [0,5] +1
3) Update [3,5] +1
4) Update [4,7] -2
5) Range [3,6]
6) Range [0,2]
SEGMENT - TREE
LAZY - ARRAY

```
void updateRange(idx, left, right, qL, qR, val){
    // resolve lazy if it exists
    if(lazy[idx] != 0){
        tree[idx] += lazy[idx];
        if(left != right){
            lazy[2*idx + 1] += lazy[idx];
            lazy[2*idx + 2] += lazy[idx];
        }
        lazy[idx] = 0;
    }
    // No overlap
    if(left > qR || right < qL) return;

    // Complete Overlap
    if(ql <= left && right <= qR){
        tree[idx] += val;
        if(left != right){
            lazy[2*idx + 1] += val;
            lazy[2*idx + 2] += val;
        }
        return;
    }
    // Partial overlap
    int mid = (left + right)/ 2;
    updateRange(lc, left, mid, qL, qR, val);
    updateRange(rc, mid+1, right, qL, qR, val);
    tree[idx] = min(tree[lc], tree[rc]);
    return;
}
```

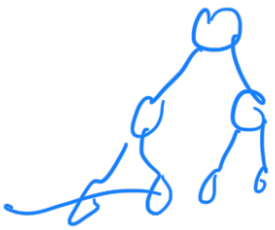
[qL, qR], val

T.C

1) No-overlap → Returning
2) Complete Overlap → Returning
3) Partial Overlap → Recur for left & right

Range Min / Sum

T.C : O(log n)

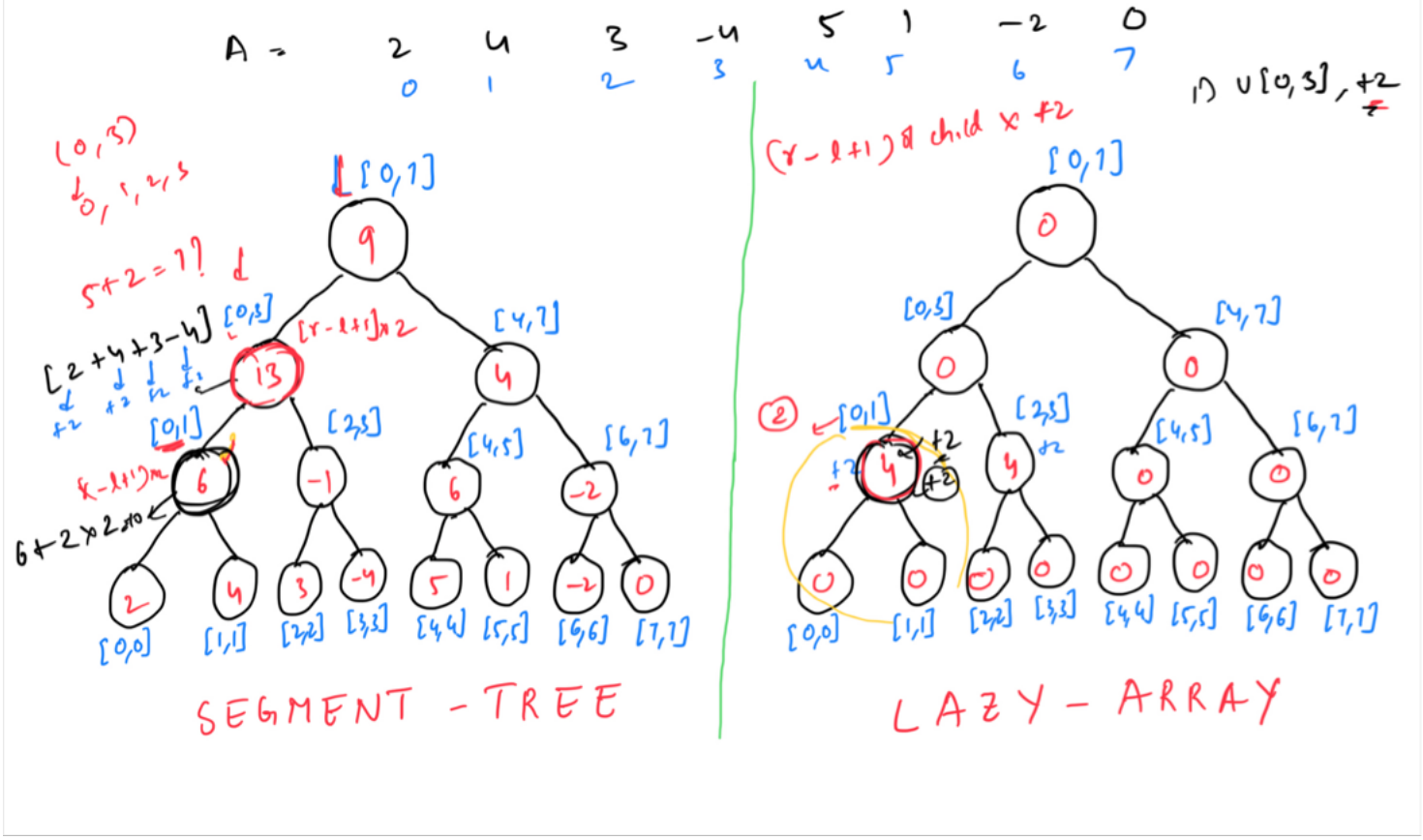


```
int queryRange(idx, left, right, qL, qR, val){
      // resolve lazy if it exists
      if(lazy[idx] != 0){
            tree[idx] += lazy[idx];
            if(left != right){
                  lazy[2*idx + 1] += lazy[idx];
                  lazy[2*idx + 2] += lazy[idx];
            }
            lazy[idx] = 0;
      }
      // No overlap
      if(left > qR || right < qL) return INF;

      // Complete Overlap
      if(s >= qL && e <= qR){
            return tree[idx];
      }

      // Partial overlap
      int mid = (s + e)/ 2;
      left = queryRange(lc, s, mid, qL, qR, val);
      right = queryRange(lc, s, mid, qL, qR, val);
      tree[idx] = min(left, right);
      return;
}
```

T.C: O(log n)

min (left, right)

Extension : Sum Segment Tree


A = 2 4 3 -4 5 1 -2 0
0 1 2 3 4 5 6 7
SEGMENT - TREE
LAZY - ARRAY

1) What should be stored at nodes of seg tree?

2) How to update parent using children

3) What to store in lazy Array

4) How to propagate Lazy value to the children.

Question: Mask Updates

Given a binary string of Length N with 0's initially

N = 5

S = " 0 0 0 0 0 "
    0 1 2 3 4

Queries

1) Range Set Bits [L, R] → No. of set bits in the range.

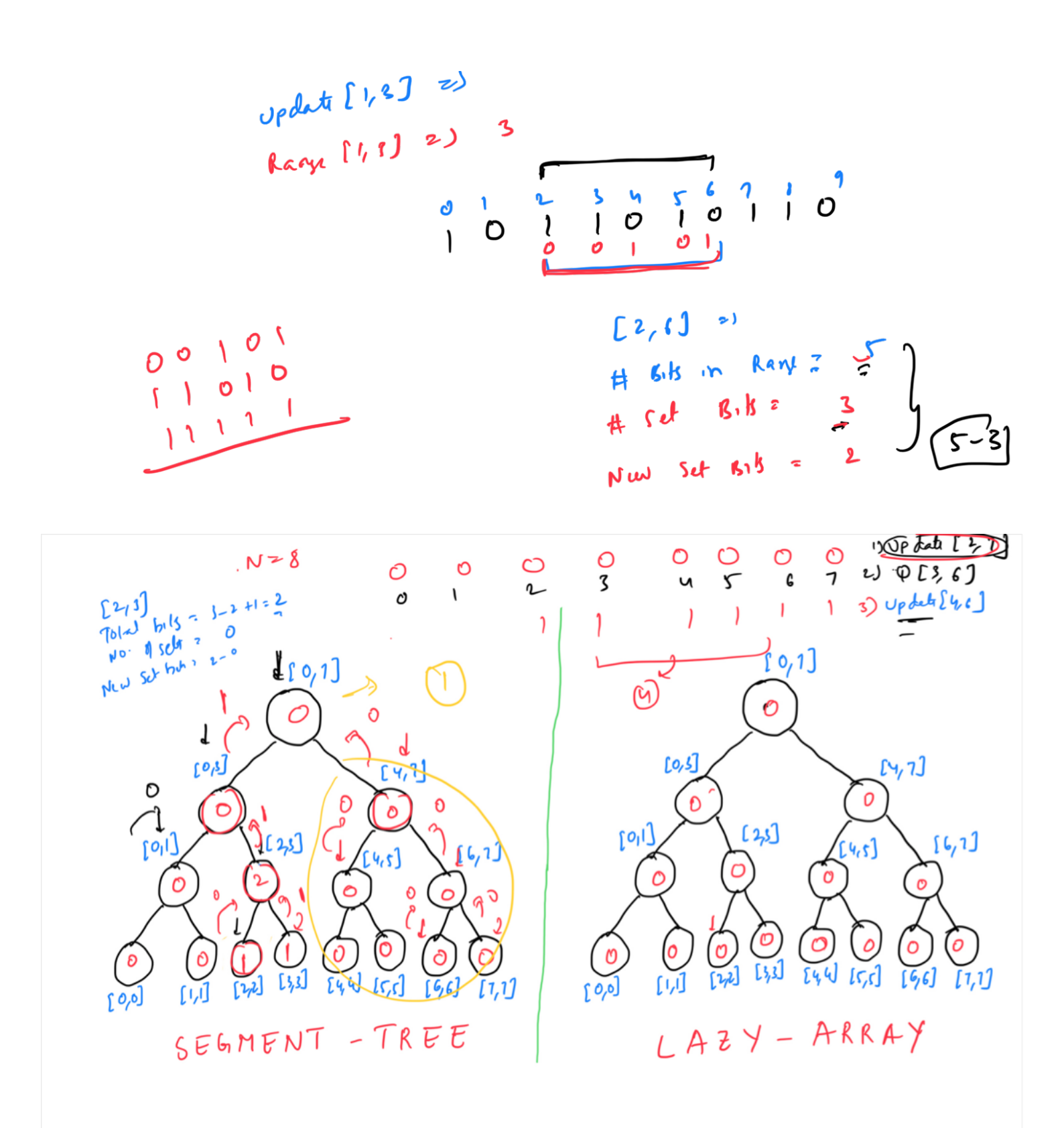
2) Range Update [L, R] → XOR all bits in this range with 1

0 ^ 1 => 1
1 ^ 1 => 0

} Flipping the bits.

S = 0 1 1 1 0
    0 1 2 3 4

Range [1, 3] => 0


Update [1,3] =>
Range [1,8] => 3
[2,6] =>
Bits in Range = 5
Set Bits = 3
New Set Bits = 2
5-3
N=8
[2,3]
Total bits = 3-2+1 = 2
No. of set = 0
New set bits = 2-0
[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
[6,7]
[0,0]
[1,1]
[2,2]
[3,3]
[4,4]
[5,5]
[6,6]
[7,7]
SEGMENT - TREE
1) Update [2,3]
2) Q [3,6]
3) Update [4,6]
LAZY - ARRAY

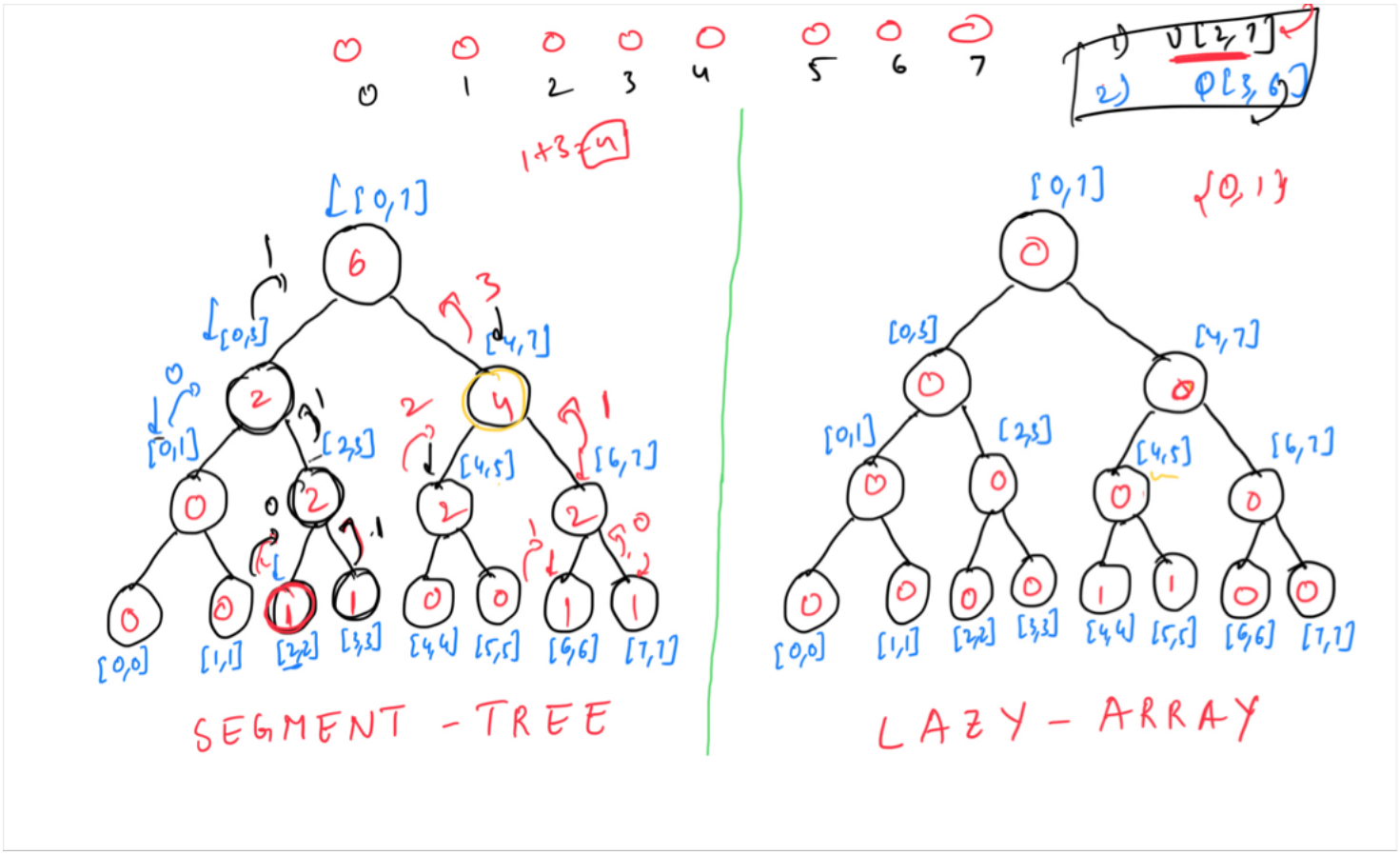
1) what to store in seg tree nodes?

Store no. of set bits in that range

2) Parent val?

Sum of 2 children

3) What to store in Lazy array?



(l, r) → [XOR → No change.

Even XOR operation → No change

odd XOR operation → Flip Operation

Circular Queries Map

N = 4

Map, Map

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

[l, R]

l ≤ R

Sum [5, 2]

2 log n

if (L > R) {

Range 1 = [L, N-1]

Range 2 = [0, R]

}

Question: Sum Segment Tree?

1) Range Sum [l, r] → Returns sum of Range from [L, R]

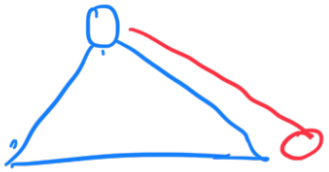
2) Update (ind, val) → A[ind] = val

3) Append (val) → Append a new value at the end of array

4) Delete (ind) → Delete element at that Index

Append

Array :



Approach 1:

Create a new array of size N+1

New array

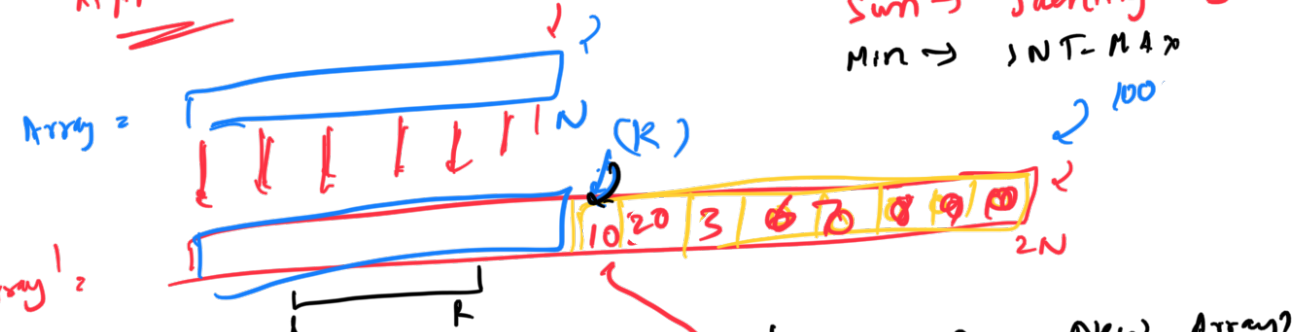
Build a Seg Tree

T.C: O(n) + O(n) => O(n)

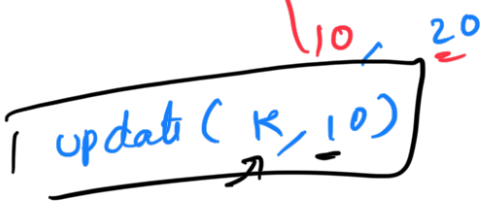
Approach 2:

Sum → Identity Size = 0

Min → INT_MAX



Build a seg tree on New Array?



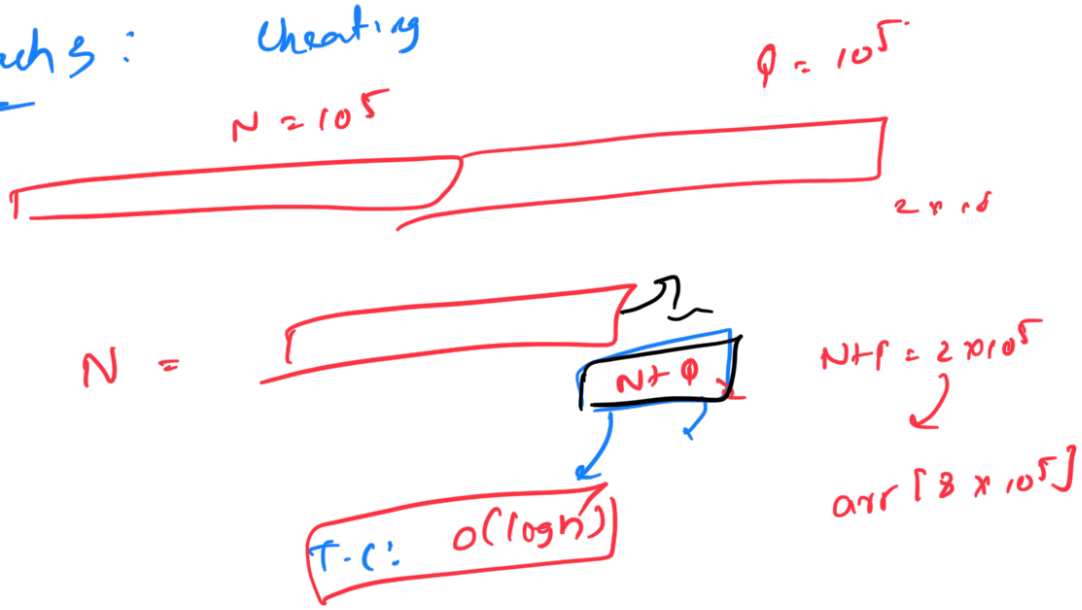
T.C: O(logn) Amortized

O(n)+O(logn) → 1 operation

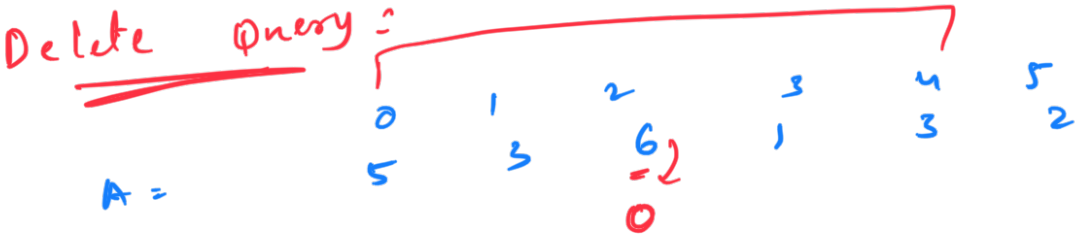
O(logn) → n →

n(logn) Amort

Approach 3 : Cheating



Delete Query :



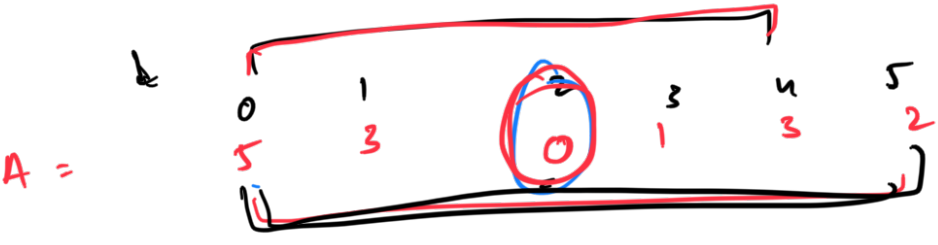
sum(0, 4) = 18 [0, 4]

Delete(2) :

Approach 1 :

1) Delete ith element from array : O(n)
2) Rebuild the segment Tree : O(n)

T.C: O(n) 5+3+1+3 [0, 4] ? [0, 5]



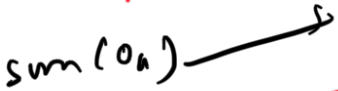
sum(0, 4) = 18

Del(2) = 5 + 3 + 0 + 1 + 3 = 12

[0, 4]

5+3+1+3+2 =

Actual = 5+3+1+3+2 = 14



of the original array

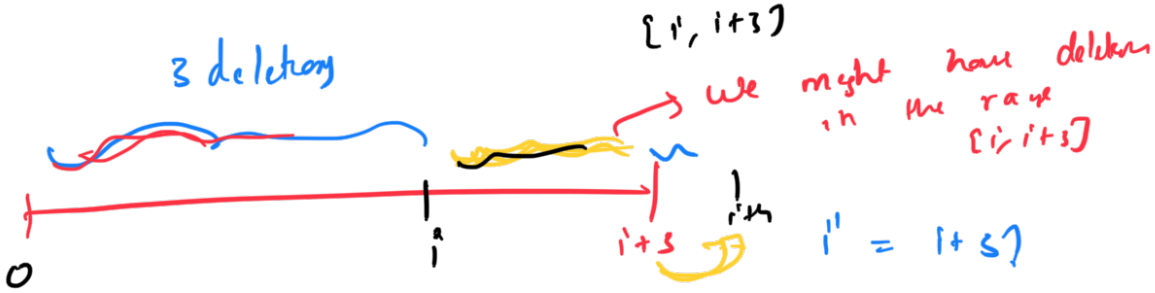
How to map the given range of new array after deletions to range of the original array

sum(0... i)

TASK: Map i to i' where i' is the index of the original array

sum(0, 4) →

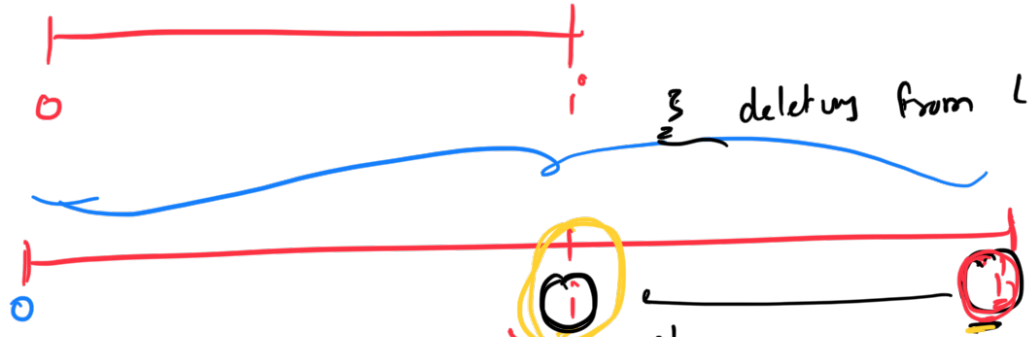
i → i' = 5



[0, i] ⇒ [0, i+3] ✗

---

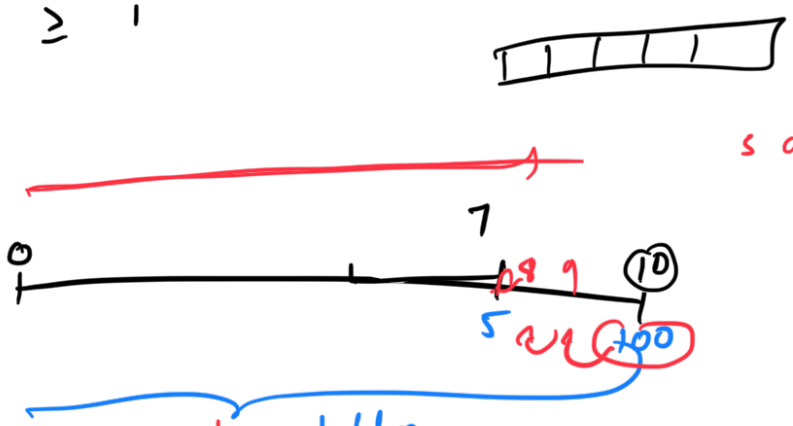
sum(0, i) → [0, i']



i' - 3 i' - 3

1) i' ≥ i

2)



$i' - \text{num\_deletions}(0 \ldots i') = i$

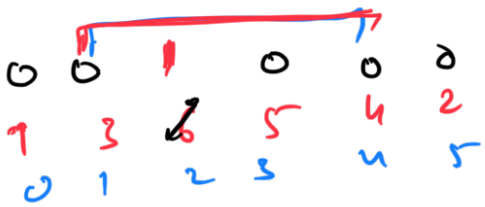
Index of i' in the new array after deletion

$i' - \text{num\_deletion}(0 \ldots i') = i$

1) No of deletions in a given range [L, R]

2) There are some deletions happens

Segment Tree



~ num_deletions(1, 4) = 0

Delete(2)

num_deletions(1, 4) = 1

Create a new array del[] initialized with 0's

del[n] = {0}

del[i] = 1 if A[i] is deleted

not deleted

del[i] = 0 if A[i] is deleted

Build a segment tree on A[i]

[l, R) → Range sum Query on Del Segment Tree

Delete(i) → del[i] = 1; Update Query

---

i - num_delete(0 ... i') = i'

	0	1	2	3	4	5	6	7
A =	5	3	0	1	0	2	4	9

Delete(2),
Delete(3),

	0	1	2	3	4	5
A' =	5	3	1	2	4	9

	5	3	0	1	0	2	4	9
A =	0	1	2	3	4	5	6	7

sum(0 ... 3) = 5+3+1+2 = 11

[0, 3] → [0, 5]

Map 3 → 5

i = 3

Find i' such that

i' - num_deleted(0, i') = 3

→ [3, 7]

$i' \geq i$ $\quad i' \rightarrow [i, n-1]$

0	1	2	3	4	5	6	7
5	3	0	1	0	2	4	9

A =

$[i, \ldots n-1]$

$i' = 7 \Rightarrow 7 - 2 = 5 > i$

$i' = 8, 9, 10, 11$

high = mid - 1

$i' = 3$ $\quad 3 - \text{num\_del}(0 \ldots 3)$

$3 - 1 = 2 < i$

$i' = 3, 2, 1$

low = mid+1

0	1	2	3
5	3	6	1

A =

A' =

0	1	2	3	4	5	6	7
5	3	6	1	4	4	4	9

-> Del(4)

-> Del(4)

$i' = 3, 4, 5$

sum $[0 \ldots 3) = 5+3+7+1 = 15$

$[0, 3] \rightarrow [0, 3]$

$i' = [3, 7]$

Range of

$i' = 4 \quad = 4 - \text{num\_del}(0 \ldots 4)$

$= 4 - 1 = 3$

$i = 3$

$i' = 3$

$i' = 5 \quad 5 - 2 = 3$

```
if ( i' - num_del(0 ... i') == k ) {
    ans = i';
    high = mid-1;
```

high = ...

}

low = i; high = n-1;

while (low ≤ high) {

mid = (high + low) / 2;

if(mid - num_del (0, mid) == i){

ans = mid

high = mid - 1;

} else if(mid - num_del (0, mid) > i),

high = mid-1;

else low = mid + 1;

Forward Pass

}

length of search space : N

B.S Iterations ⇒ O(log n)

T.T.C : O(log n * log n)

1) Append

Arr [n] = ele

Update (n, ele);

n++;

N is the no. of elements so far

} O(log n)

2) Update : update (ind, val)

→

i' = Find Index (ind)

update (i', val);

} (log n)² + log n

= O((log n)²)

Delete : Delete (ind)

(log n)²

3) Delete

i' = findIndex(ind);

update(i', 0);

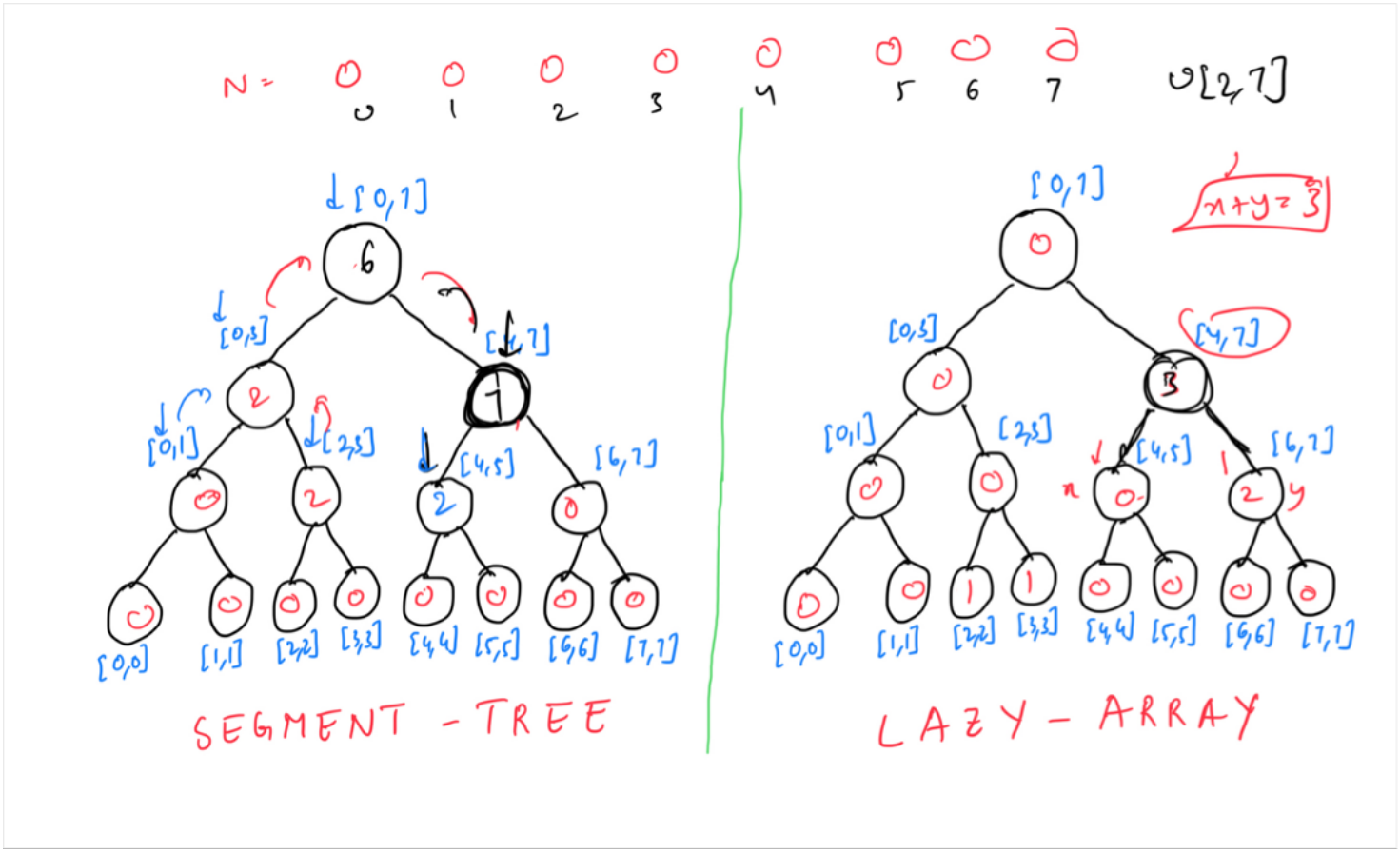
// del[i'] = 1 and Update in segment tree

4) Range Query : (L, R)

L' = findInd(L);

R' = findInd(R);

rangeSum(L', R');



```
void updateRange(idx, left, right, qL, qR, val){
      // resolve lazy if it exists
      if(lazy[idx] != 0){
            tree[idx] += lazy[idx];
            if(left != right){
                  lazy[2*idx + 1] += lazy[idx];
                  lazy[2*idx + 2] += lazy[idx];
            }
            lazy[idx] = 0;
      }
      // No overlap
      if(left > qR || right < qL) return;

      // Complete Overlap
      if(ql <= left && right <= qR){
            tree[idx] += val;
            if(left != right){
                  lazy[2*idx + 1] += val;
                  lazy[2*idx + 2] += val;
            }
            return;
      }
      // Partial overlap
      int mid = (left + right)/ 2;
      updateRange(lc, left, mid, qL, qR, val);
      updateRange(rc, mid+1, right, qL, qR, val);
      tree[idx] = min(tree[lc], tree[rc]);
      return;
}
```

```
int queryRange(idx, left, right, qL, qR, val){
      // resolve lazy if it exists
      if(lazy[idx] != 0){
            tree[idx] += lazy[idx];
            if(left != right){
                  lazy[2*idx + 1] += lazy[idx];
                  lazy[2*idx + 2] += lazy[idx];
            }
            lazy[idx] = 0;
      }
      // No overlap
      if(left > qR || right < qL) return INF;

      // Complete Overlap
      if(s >= qL && e <= qR){
            return tree[idx];
      }

      // Partial overlap
      int mid = (s + e)/ 2;
      left = queryRange(lc, s, mid, qL, qR, val);
      right = queryRange(lc, s, mid, qL, qR, val);
      tree[idx] = min(left, right);
      return;
}
```