Enroll
Page No: 589
Date: / /

SEGMENT TREES

* "A Segment tree is a data structure that allows efficient range queries & updates on an array."
* It is a especially useful for scenarios where we need to perform queries & updates repeatedly on a range of elements in an array.

★ WHY DO WE NEED SEGMENT TREES?

(i) Efficient Range Queries

* Without segment trees, performing range queries (like summing elements in a sub array) requires iterating over the specified range, this would take $O(n)$. If there are many queries, becomes very inefficient.
* Segment trees reduce the time to $O(\log n)$ for each query by storing precomputed range values, making it possible to retrieve range information quickly.

(ii) Efficient Updates

* If you update an element in the array & don't use a segment tree, you would need to adjust all relevant range sums in $O(n)$.
* Segment tree enables us to perform these updates in $O(\log n)$ by only adjusting the affected segments, making them efficient for arrays that are frequently updated.

Enroll
Page No: 590
Date: / /

★ COUNT & MEANING OF NODES

mid = (start + end)/2
left ⇒ (start, mid)
right ⇒ (mid+1, end)

No. of nodes

$(2^0) = 1$ ← 1, 2, 3, (4), 5, 6, 7, 8 (mid)

$(2^1) = 2$ ← 1, (2), 3, 4 (mid) ; 5, (6), 7, 8 (mid)

$(2^2) = 4$ ← (1) 2 (3) 4 ; (5) 6 (7) 8 (mid)

$(2^3) = 8$ ← 1 2 3 4 5 6 7 8

* Total nodes (for x levels) $= \underbrace{2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^x}$

G.P. (Geometric Propogation)

$\rightarrow Sum = \frac{a(r^n - 1)}{r - 1} = \frac{1(2^{x+1} - 1)}{1}$

$\rightarrow Sum = 2^{x+1} - 1$

$\Rightarrow$ **Total Nodes $= 2^{x+1} - 1$**

∴ x = last level

* We know,

$\rightarrow x$ = last level

$\frac{n}{2^x} = 1$

$n = 2^x$

$\Rightarrow x = \log_2 n$

Levels
0th → n → $n/2^0$
1st → $n/2$ → $n/2^1$
2nd → $n/4$ → $n/2^2$
3rd → $n/8$ → $n/2^3$
⋮
xth → $n/2^x$ → 1

Now, Total nodes $= 2^{x+1} - 1 = 2^{\log n + 1} - 1 = 2^{\log_2 n + \log_2 2} - 1$

$= 2^{\log_2 (2n)} - 1$

$\Rightarrow 2n - 1$

// Next Page

Enroll
Page No: 591
Date: / /

⇒ Total Levels = $\log_2 n$ ⇒ Total Nodes = $2n - 1$

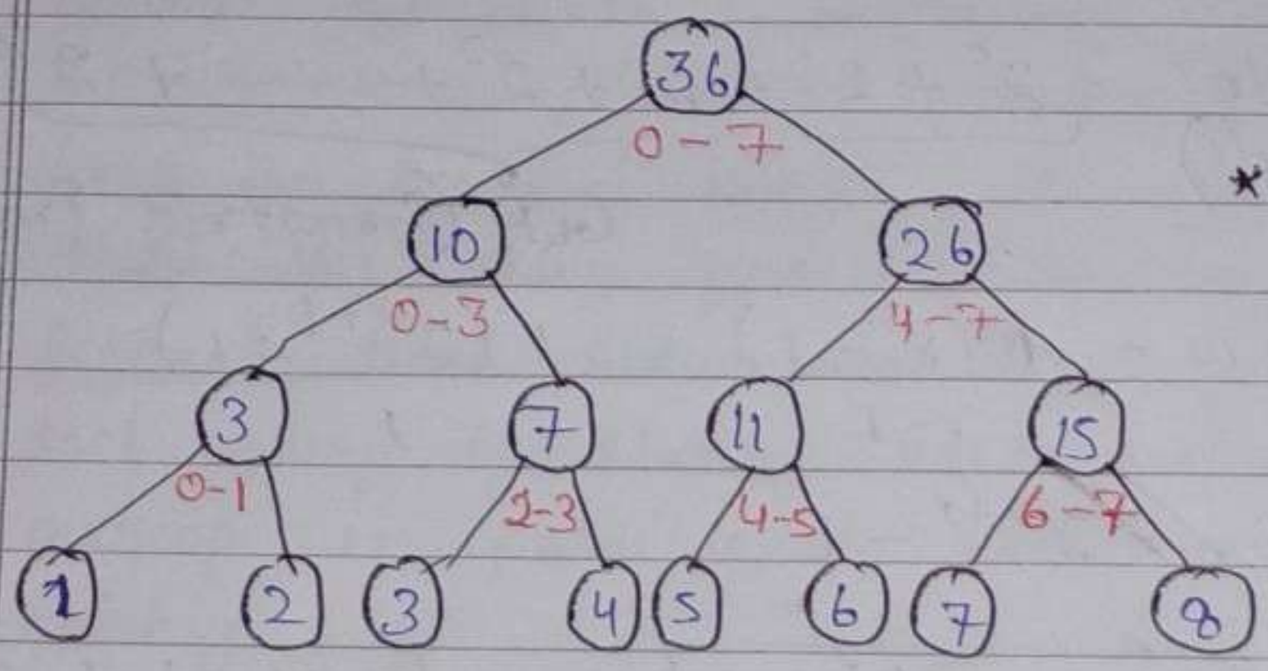
⇓

Full BT: Each nodes has 2 children ~~exp~~ except leaf.

In Code, we take $4*n$ becoz of safety that we make full binary tree so space do not come less, by the way answer will come $2n-1$ but we use $4*n$.

Basically, inside nodes we add answers of ranges.

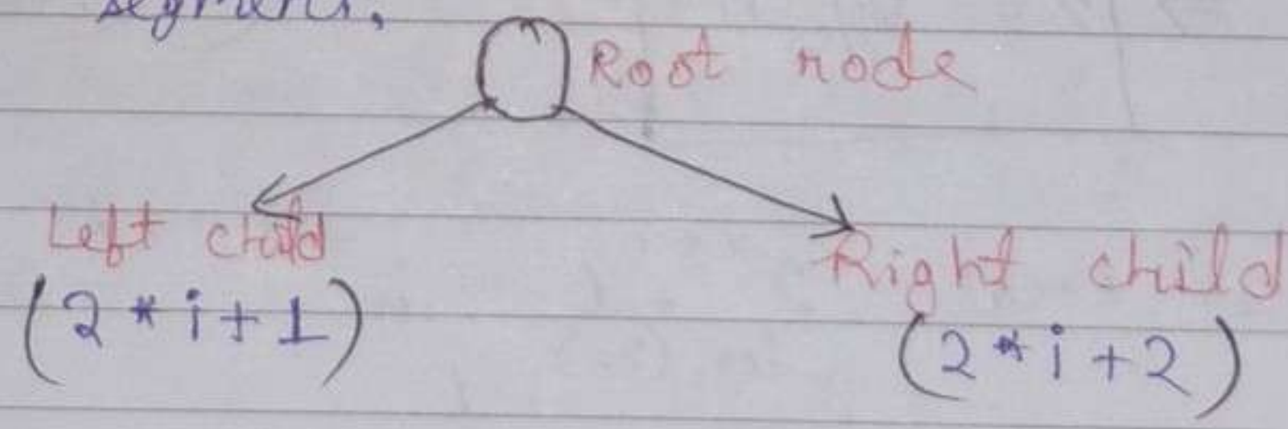
Ex:- At page 590, given array subarray's sum using segment tree.



* nodes = answer of range.

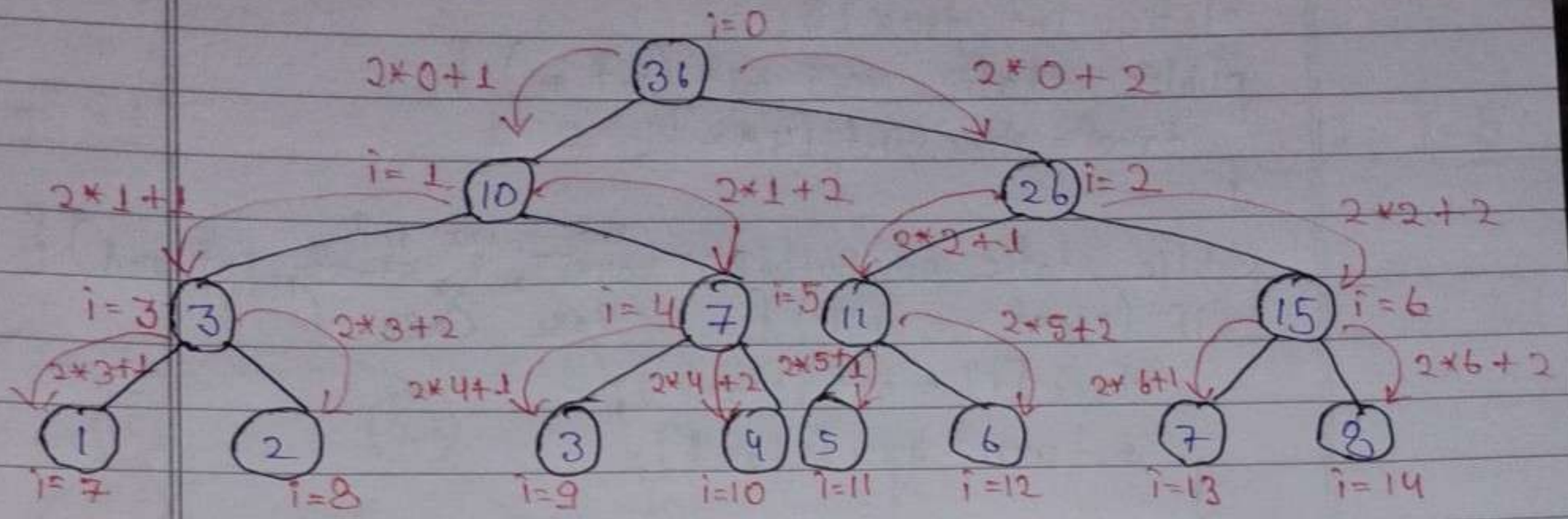
★ CREATION OF SEGMENT TREE

* We can represent segment tree as an array of size $4*n$ i.e., tree[] where each node at index i in the tree array stores information for a segment.



Enroll
Page No: 592
Date: / /

Ex→ For Sum of Subarray, → start 1, 2, 3, 4, 5, 6, 7, 8 end



Now, → tree[4*n] =

36	10	26	3	7	11	15	1	2	3	4	5	6	7	8	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	32

i ⇒

* The empty indexes will be filled with zero (0).

Pseudo Code →

```
tree [ ] = 4*n size i.e., 4*8 = 32.
buildST(arr, i, start, end){
 //Base case
 if (start == end)
     ~~prin~~ tree[i] = arr[start]
 mid = (start + end) / 2
Left = (buildST(arr, 2*i+1, start, mid))
right = (buildST(arr, 2*i+2; mid+1, end))

 tree [i] = ~~(2+)~~ left + right
 return tree [1];
}
```

Output: 36 10 26 3 7 11 15 1 2 3 4 5 6 7 8 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Enroll
Page No: 593
Date: / /

⇒
```
public class CreationOfST {
    static int tree [];
    public static void init (int n) {
        tree = new int [4 * n];
    }
    public static int buildBST (int arr[], int i, int start, int end) {
        if (start == end) { // Base Case
            tree [i] = arr [start];
            return arr [start];
        }
        int mid = (start + end) / 2;
        buildBST (arr, 2*i+1, start, mid); //Left subtree
        buildST (arr, 2*i+2, mid+1, end); //right subtree
        tree[i] = tree [2*i+1] + tree [2*i+2];
        return tree [i];
    }
    public static void main (String args []) {
        int arr [] = {1, 2, 3, 4, 5, 6, 7, 8};
        int n = arr.length;
        init (n);
        buildST (arr, 0, 0, n-1);
        for (int i=0; i<tree.length; i++) {
            System.out.print (tree [i] + " ");
        }
    }
}
```

Time Complexity

* To Construct ST — $O(n)$
* For Query on ST — $O(\log n)$
* For Update on ST — $O(\log n)$

Enroll
Page No: 594
Date: / /

★ QUERY ON SEGMENT TREE
- Sum of numbers in a range

Input:

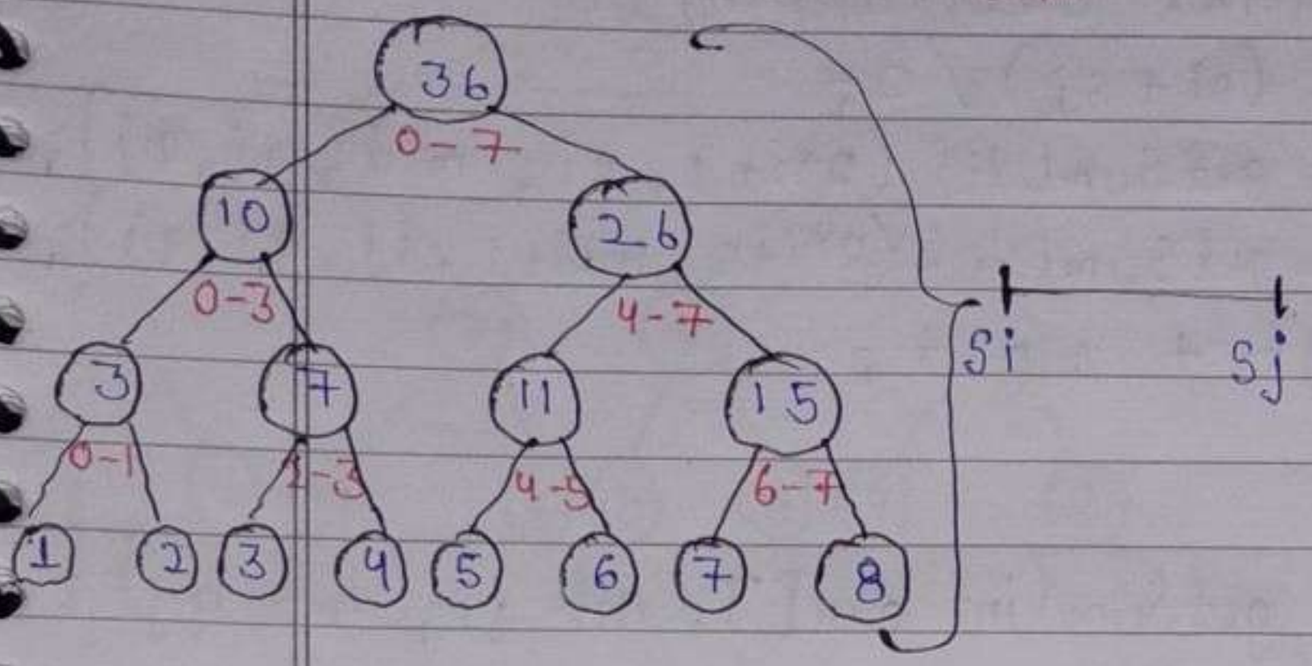
0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

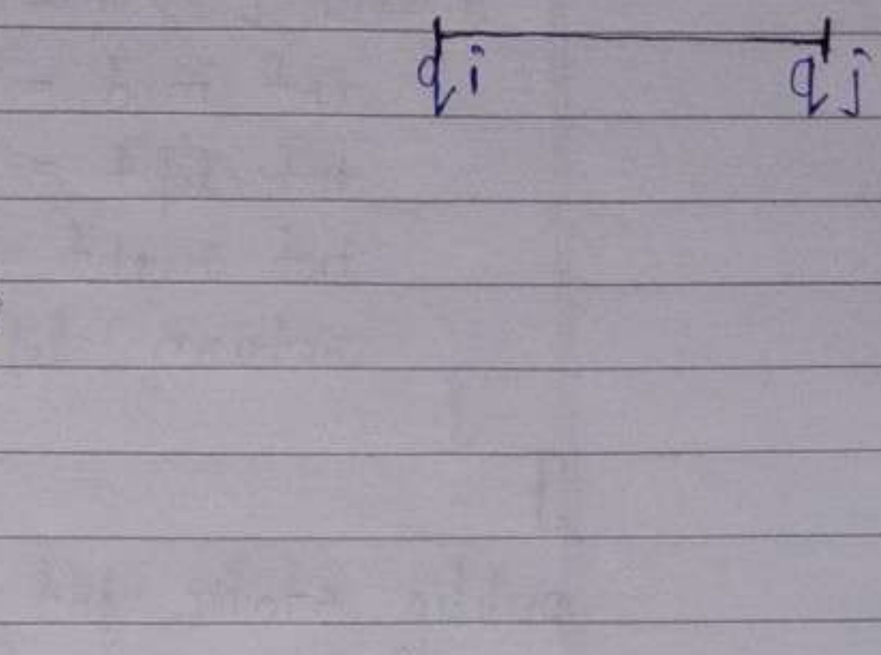
i

j

sum

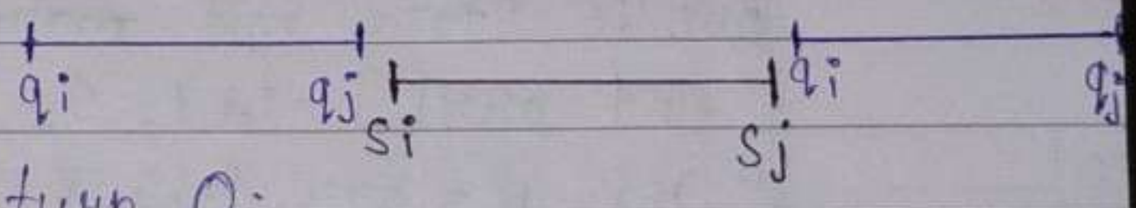
i=2, i=5



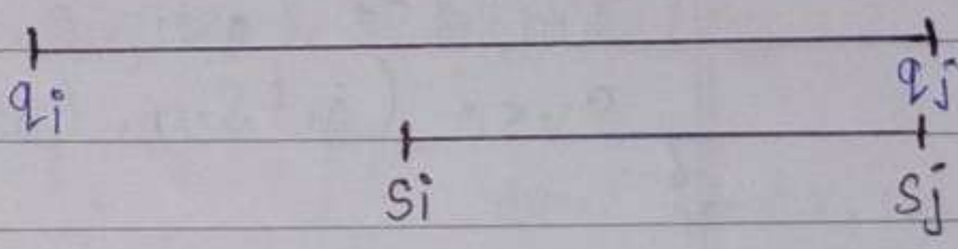


* There will be 3 cases that are :-

Case 1: Non-Overlapping
(qj <= si) || (qi >= sj)
not include. ⇒ return 0;



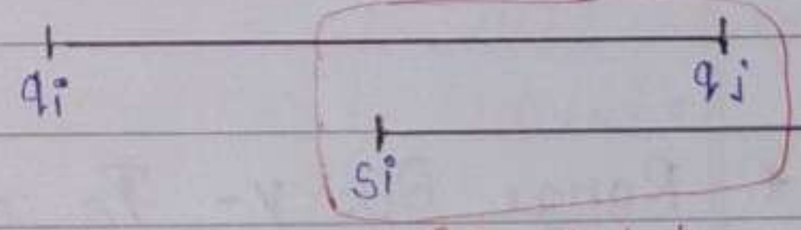
Case 2: Complete Overlapping
(si >= qi && sj <= qj)
include tree [i]



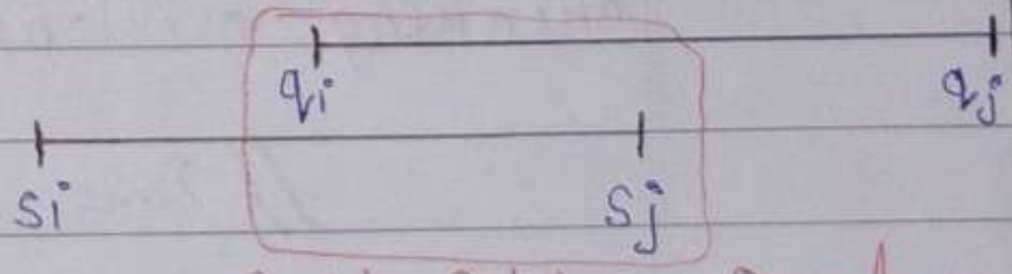
Case 3: Partial Overlapping
mid = (si + sj) / 2

Left ⇒ (si, mid)
+
right ⇒ (mid+1, j)

↓
return sum;



Left Subtree Overlap



Right Subtree Overlap

⇒
```
public static int getSumUtil(int i, int si, int sj, int qi, int qj){
  if (qj <= si || qi >= sj) { // Non-Overlapping
     return 0;
  } else if (si >= qi && sj <= qj) { // Complete Overlap
     return tree[i];
  } else { // partial Overlapping
     int mid = (si + sj) / 2;
     int left = getSumUtil(2*i+1, si, mid, qi, qj);
     int right = getSumUtil(2*i+2, mid+1, sj, qi, qj);
     return left + right;
  }
}

public static int getSum(int arr[], int qi, int qj) {
  int n = arr.length;
  return getSumUtil(0, 0, n-1, qi, qj);
}

public static void main(String args[]) {
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
  int n = arr.length;
  init(n);
  buildSt(arr, 0, 0, n-1);
  Syso(getSum(arr, 2, 5)); // 18
}
```

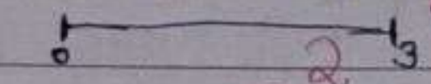
* Range Query- To get the result for a range [L, R], the tree is traversed by checking overlapping segments.

// Time Complexity - O(log n)

Enroll
Page No: 596
Date: / /

★ UPDATE ON SEGMENT TREE

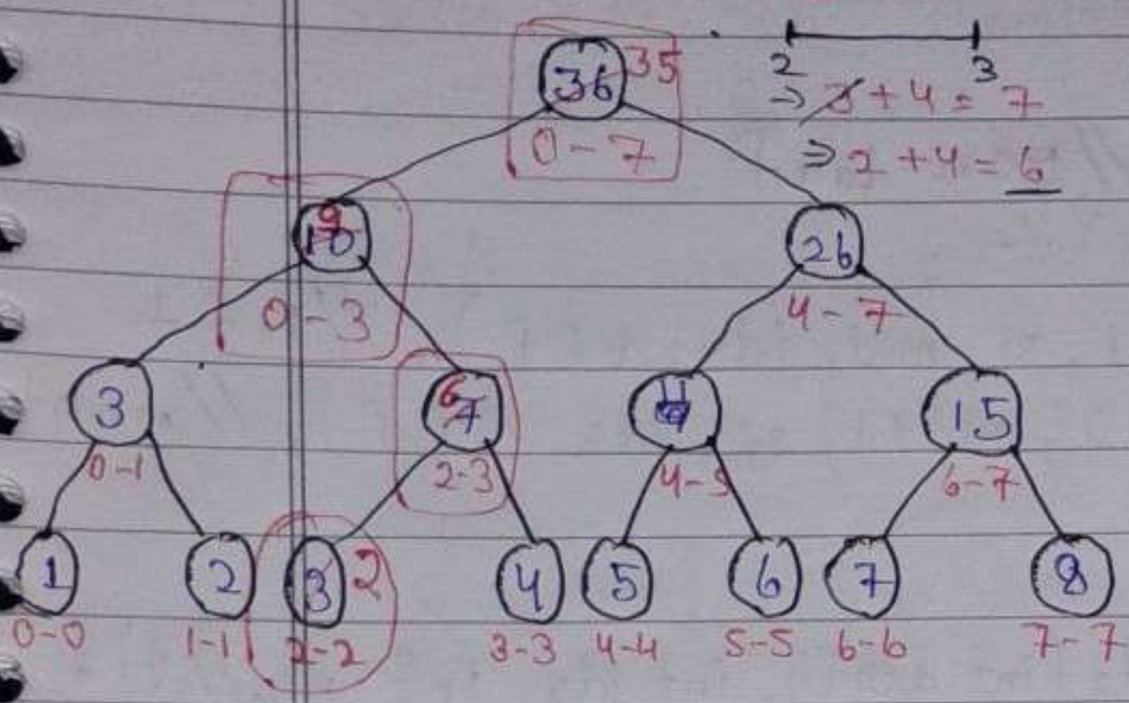
– Update at an index idx



1+2+2+4 = 9

i = 1, 2, 3, 4, 5, 6, 7, 8
0 1 2 3 4 5 6 7

idx = 2 ⇒ 2



* Before Updation:

tree[] :	36	10	26	3	7	11	15	1	2	3	4	5	6	7	8	0	0	------	0
i =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	------	32

* After Updation:

tree[] :	35	9	26	3	6	11	15	1	2	2	4	5	6	7	8	0	0	------	0
i =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	------	32

* Time Complexity for Single Updation → O(log n)

* For array, [1, 2, ~~3~~ 2, 4, 5, 6, 7, 8] } arr[], idx, newVal
diff = newVal - arr[idx]
update

* For Segment tree,

```
Update Util(i, si, sj, idx, diff)
if (idx > sj || idx < si) return;
tree[i] = diff
if (si != sj) {
    mid = (si + sj)/2
    left(2i+1, si, mid, idx, diff)
    right(2i+2, mid+1, sj, idx, diff)
}
```

⇒ // Time Complexity - O(log n)

```
public static void updateUtil(int i, int si, int sj, int idx, int diff){
    if (idx > sj || idx < si) {
        return;
    }
    tree[i] += diff;
    if (si != sj) { //non-leaf
        int mid = (si+sj)/2;
        updateUtil(2*i+1, si, mid, idx, diff); //left
        updateUtil(2*i+2, mid+1, sj, idx, diff); //right
    }
}

public static void update(int arr[], int idx, int newVal) {
    int n = arr.length;
    int diff = newVal - arr[idx];
    arr[idx] = newVal;
    updateUtil(0, 0, n-1, idx, diff); //segment tree updation
}

public static void main(String args[]) {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int n = arr.length;
    init(n);
    buildST(arr, 0, 0, n-1);
    Syso(getSum(arr, 2, 5)); //18
    update(arr, 2, 2);
    Syso(getSum(arr, 2, 5)); //17
}
```

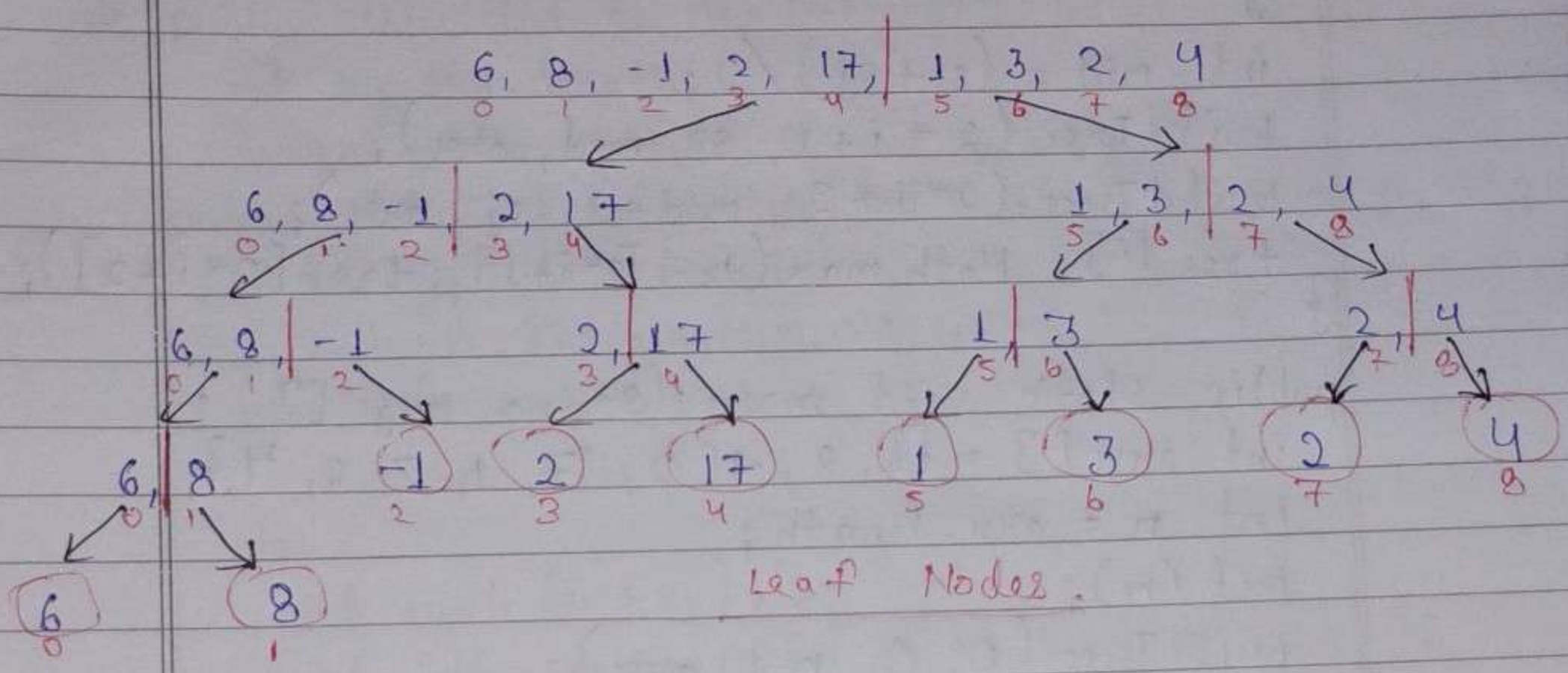
Enroll
Page No: 598
Date: / /

★ MAX ELEMENT QUERIES

→ Given an arr[], we have to answer few queries:
a) Output Max for the subarray [i...j].
b) Update the element at idx

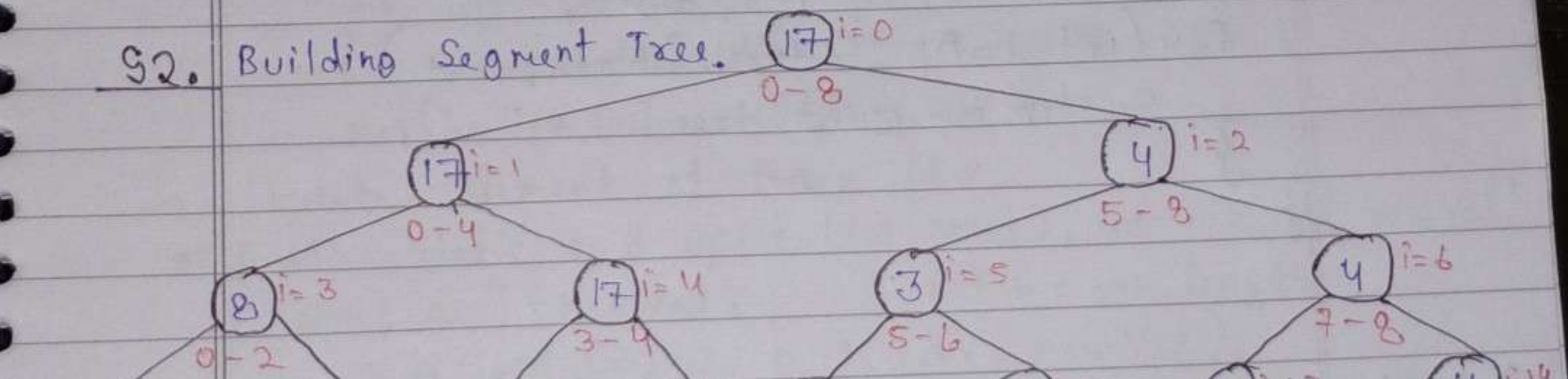
{6, 8, -1, 2, 17, 1, 3, 2, 4} → n = 9
i= 0 1 2 3 4 5 6 7 8

S1. Create a array i.e, tree[] = new int [4*n = 4*9 = 36]



Leaf Nodes.

S2. Building Segment Tree.



* We have to fill nodes with answer, i.e. Max element given b/t the given range.

tree[] ⇒

i=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	----	36
	17	17	4	8	17	3	4	8	-1	2	17	1	3	2	4	6	8	0	0	----	0

Enroll
Page No: 599
Date: / /

⇒
```
static int tree[];
public static void init (int n) {
    tree = new int [4*n];
}
public static void buildTree (int i, int si, int sj, int arr[]) {
    if (si == sj) {
        tree [i] = arr [si];
        return i
    }
    int mid = (si + sj) / 2;
    buildTree (2 * i + 1, si, mid, arr);
    buildTree (2*i + 2, mid + 1, sj, arr);
    tree [i] = Math.max (tree [2*i+1] ; tree [2*i+2]);
}
public static void main (String args []) {
    int arr [] = {6, 8, -1, 2, 17, 1, 3, 2, 4};
    int n = arr.length;
    init (n);
    buildTree (0, 0, n-1, arr);
    for (int i=0; i< tree.length; i++) {
        System.out.print (tree [i] + " ");
    }
}
```

Output:
```
17 17 4 8 17 3 4 8 -1 2 17 1 3 2 4
6 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

Enroll
Page No: 600
Date: / /

★ MIN ELEMEN

a) Output Max for subarray [i..j].

C1: No Overlap ⟹ return $-\infty$;

C2: ~~Ov~~ Complete Overlap ⟹ return tree [i];

C3: Partial Overlap ⟹
mid = (si + ei) / 2
Left (2i+1, si, mid, qi, qj)
right (2i+2, mid+1, sj, qi, qj)
return max (Left, right).

⇒ public static int getMax (int arr[], int qi, int qj)
int n = arr.length;
return getMaxUtil (0, 0, n-1, qi, qj);
public static int getMaxUtil (int i, int si, int sj, int qi, int qj){
if (si > qj || sj < qi)
return Integer.MIN_VALUE;
else if (si >= qi && sj <= qj)
return tree [i];
else
int mid = (si + sj) / 2;
int left = getMaxUtil (2*i+1, si, mid, qi, qj);
int right = getMaxUtil (2*i+2, mid+1, sj, qi, qj);
return Math.max (left, right);

b) Update element at index idx
⇒ public static void update (int arr[], int idx, int newVal)
arr [idx] = newVal;
Update Util (0, 0, n-1, idx, newVal);
public static void updateUtil (int i, int si, int sj, int idx, int newVal)
if (idx < si || idx > sj)
return;
tree [i] = Math.max (tree [i], newVal);
if (si != sj)
int mid = (si + sj) / 2;
updateUtil (2*i+1, si, mid, idx, newVal); // left
updateUtil (2*i+2, mid+1, sj, idx, newVal); // right

int n = arr.length;

THE END

"After months of diving into DSA, I've filled 600 pages of knowledge, yet the real mastery of Java & DSA still lies Ahead."

→ This is only the foundation of what's yet to come!