**Report Segment Tree (Seminar)**

**1/Introduction:**

The segment tree was invented by Jon Bentley in 1977; in "Solutions to Klee’s rectangle problems"

Segment Tree is a basically a binary tree used for storing the intervals or segments. Each node in the Segment Tree represents an interval. Consider an array A of size N and a corresponding Segment Tree T:

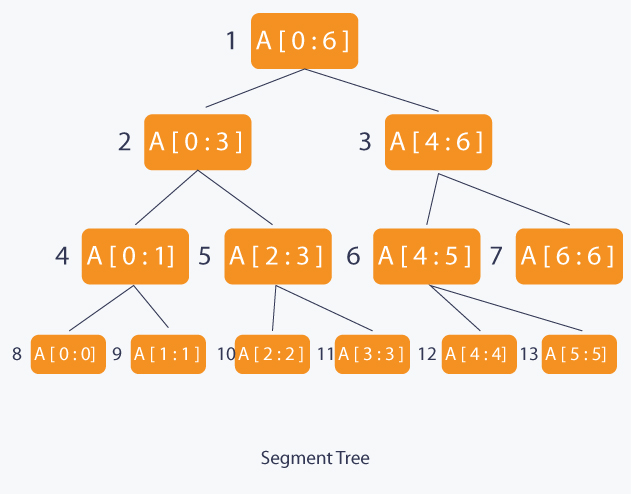
1. The root of T will represent the whole array A[0:N−1].
2. Each leaf in the Segment Tree T will represent a single element A[i] such that 0≤i<N.
3. The internal nodes in the Segment Tree T represents the union of elementary intervals A[i:j] where 0≤i<j<N.

The root of the Segment Tree represents the whole array A[0:N−1]. Then it is broken down into two half intervals or segments and the two children of the root in turn represent the A[0:(N−1)/2] and A[(N−1)/2+1:(N−1)]. So in each step, the segment is divided into half and the two children represent those two halves. So the height of the segment tree will be log2N. There are N leaves representing the N elements of the array. The number of internal nodes is N−1. So, a total number of nodes are 2×N−1.

Once the Segment Tree is built, its structure cannot be changed. We can update the values of nodes but we cannot change its structure. Segment tree provides two operations:

1. **Update:** To update the element of the array A and reflect the corresponding change in the Segment tree.
2. **Query:** In this operation we can query on an interval or segment and return the answer to the problem (say minimum/maximum/summation in the particular segment).

The Segment Tree of array A of size 7 will look like :



**-***A data structure for range query processing*

*-Divide and conquer approach*

*- Complete binary tree*

*- Each node is responsible for query of consecutive elements in [l,r]*

**2/Problem:**

-Let A be an array with M elements (M < 105) , N queries (N < 105). There are two kinds of queries:

We set value v with the element at position i

Find the maximum value on the interval [i ; j]

The easy way that we can use is using an array A to store elements. With step i we can set A[i] = v. With step ii we can use a loop from i to j to find the maximum element in the interval [i ; j]. But this approach is not efficiency because the complexity of full algorithm is and run- time is out when M and N are too large.

So we can use the segment tree to minimize the complexity and run – time when we solve this problem. With the approach of segment tree, we can do as follow:

(\*)With the query 1, we can set up the value of the Node

(\*)With the query 2, we consider all nodes on the segment tree that it is on interval [i; j] and find the maximum of these nodes.

**How to construction it?**

-First, we will start from the root.

-Traverse in post order.

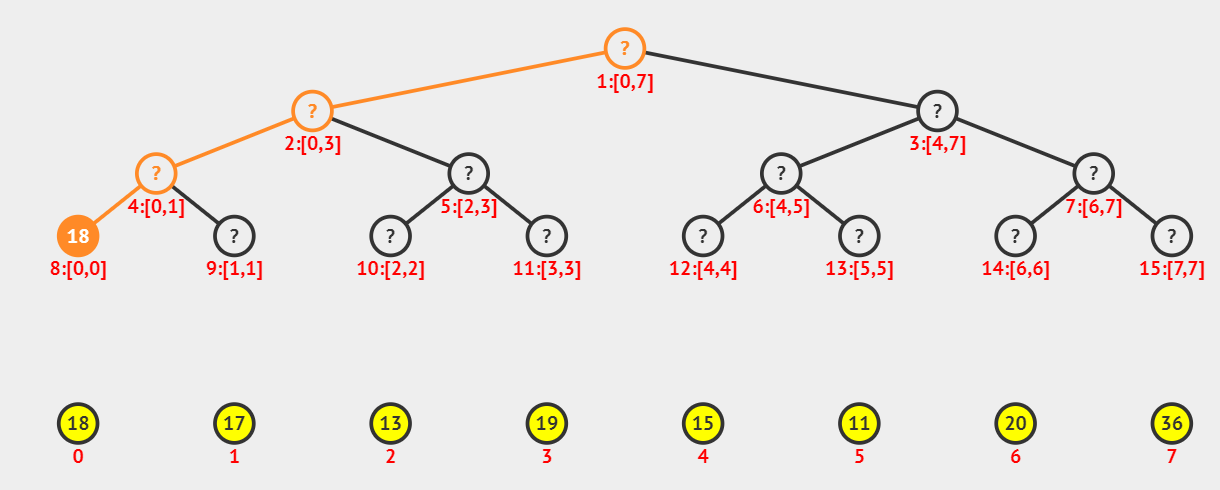
-If a is leaf node, we initialize value for node a and return.

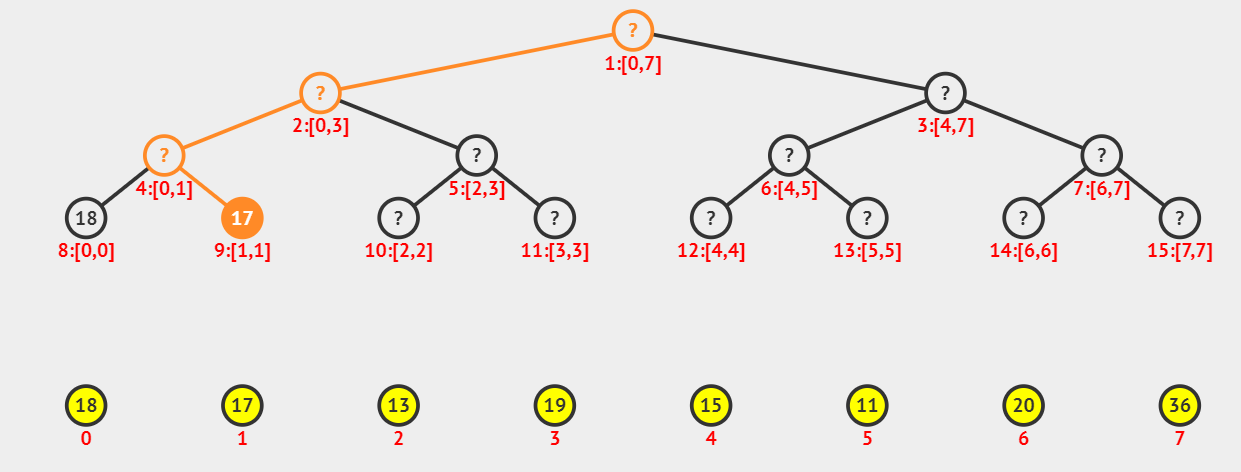
-Else if a is not a leaf node, we will recursive traverse to children nodes and update the values base on the results of children node.

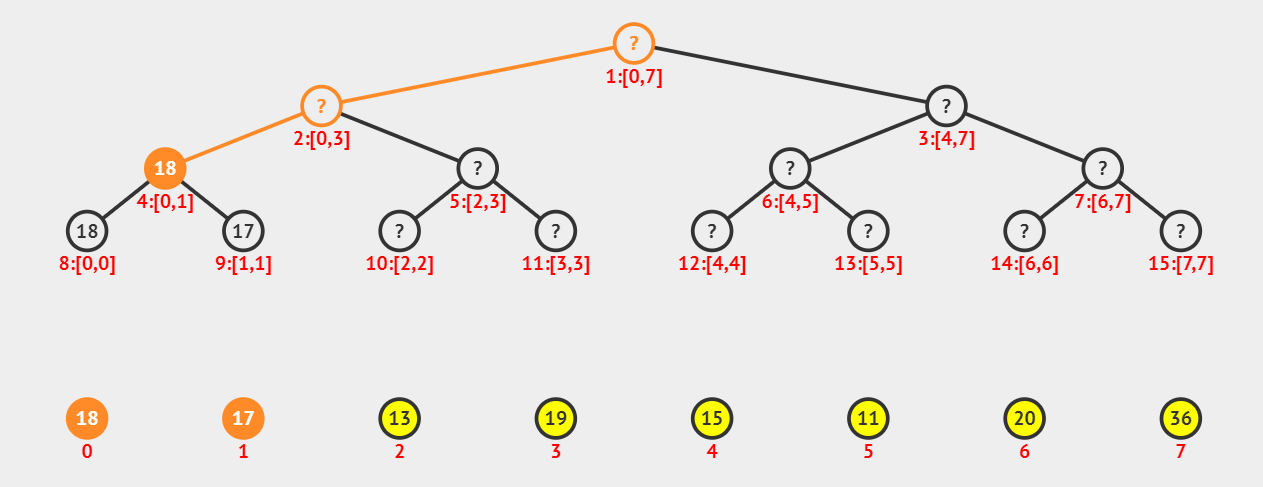
Complexity: O(nlog(n))

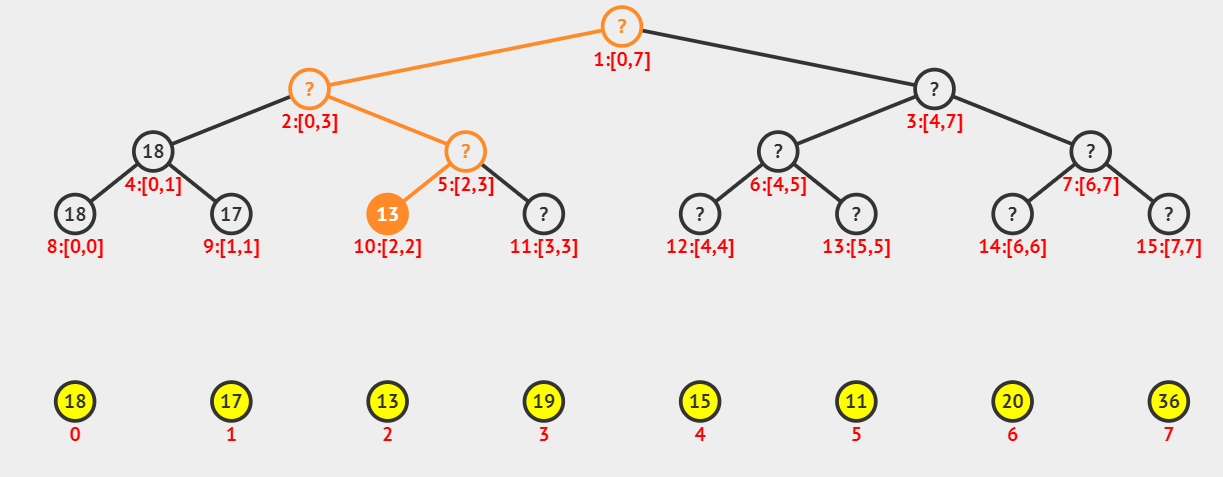
Return to the problem, we consider N = 8 and set

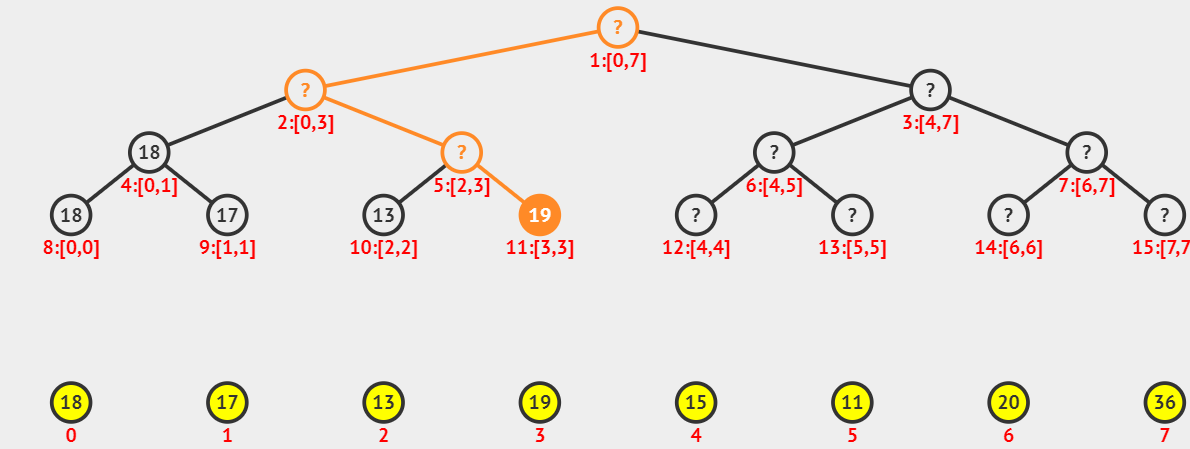
Link: <https://visualgo.net/en/segmenttree>

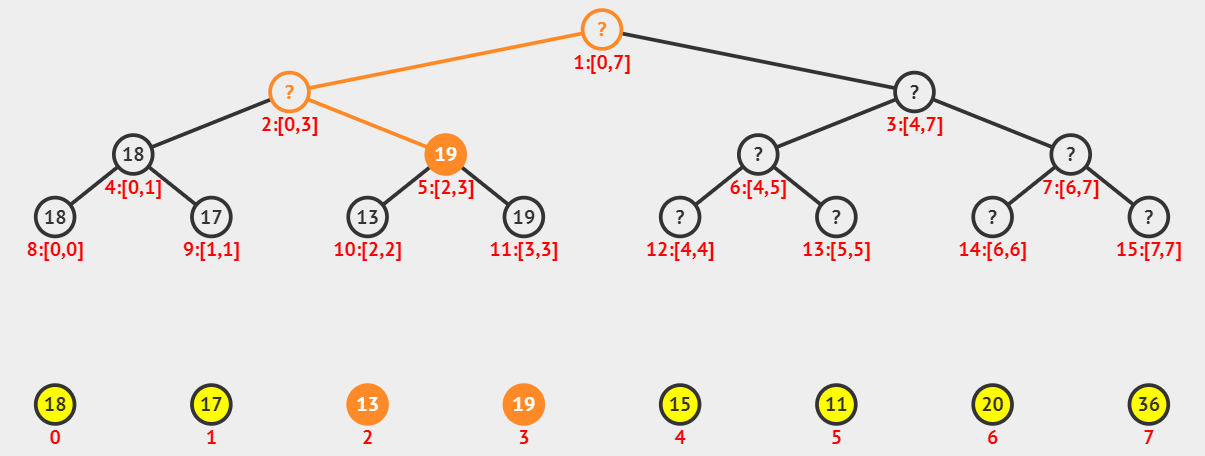


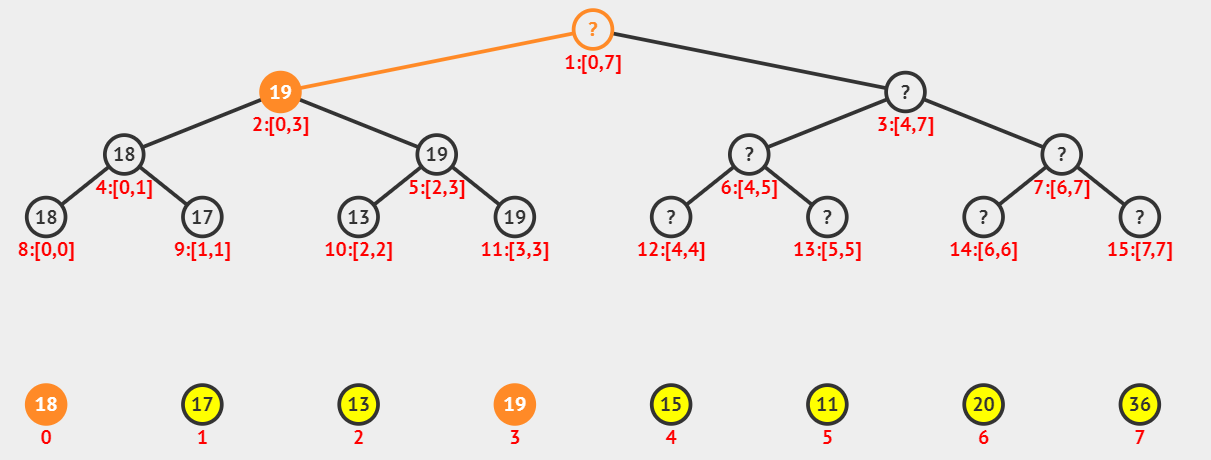


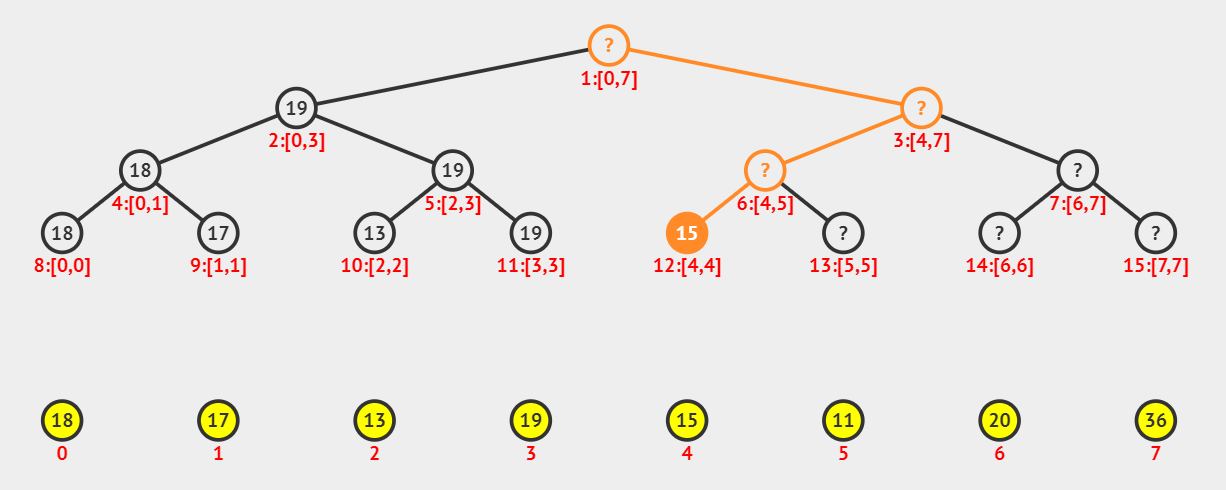


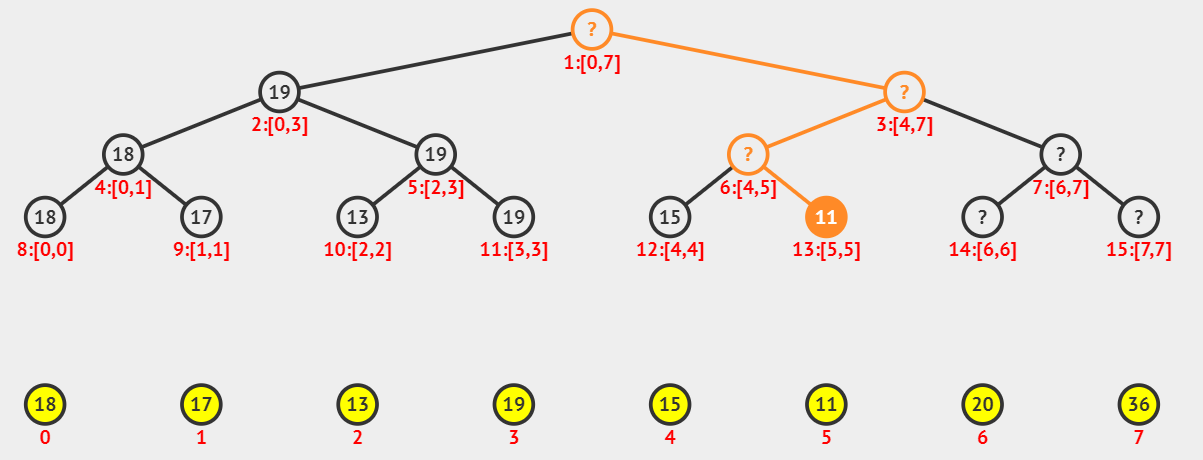


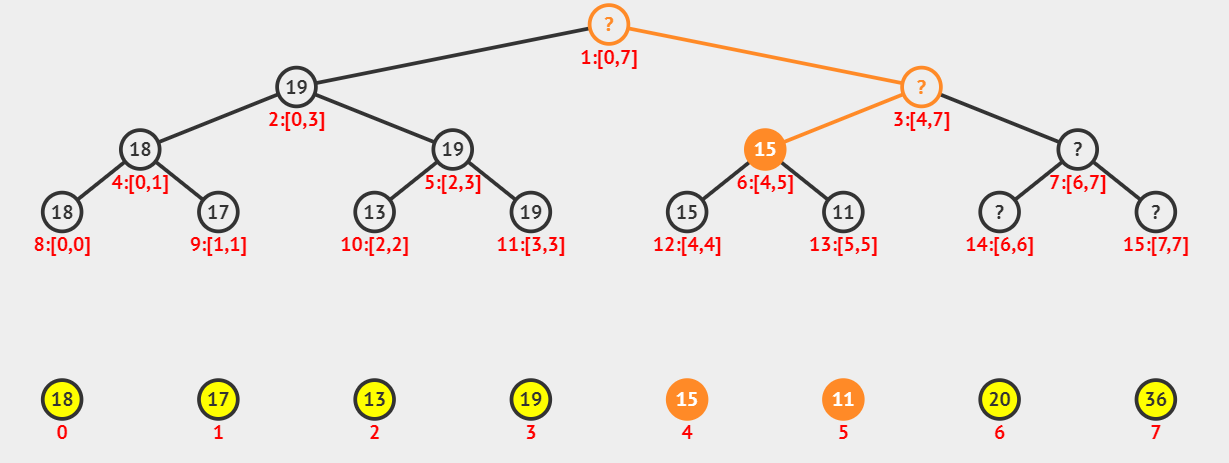


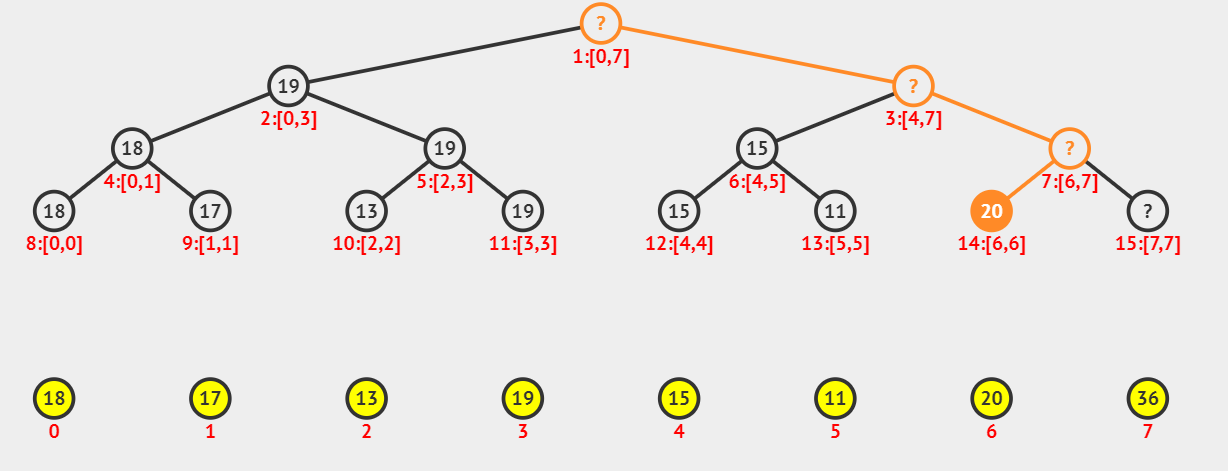


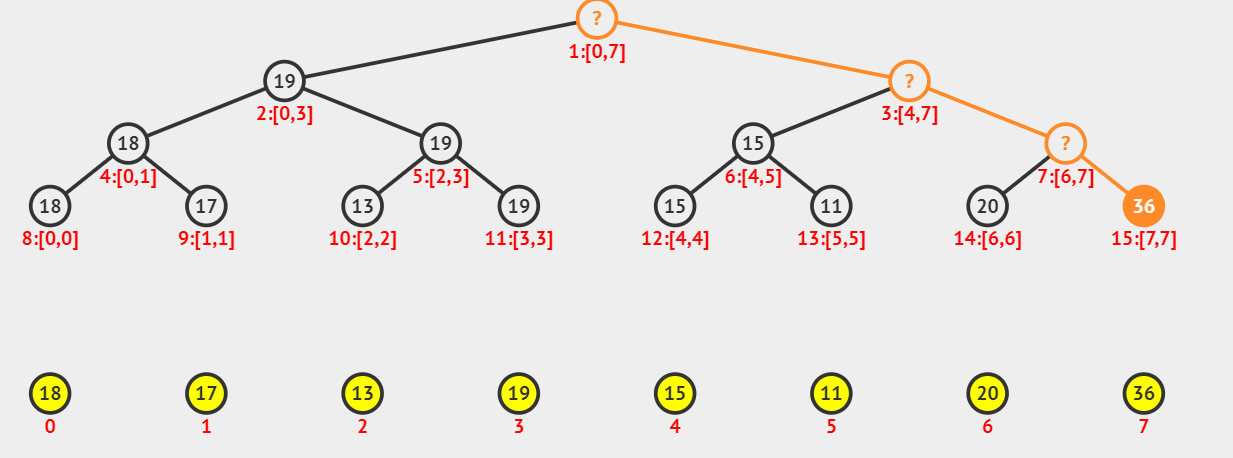


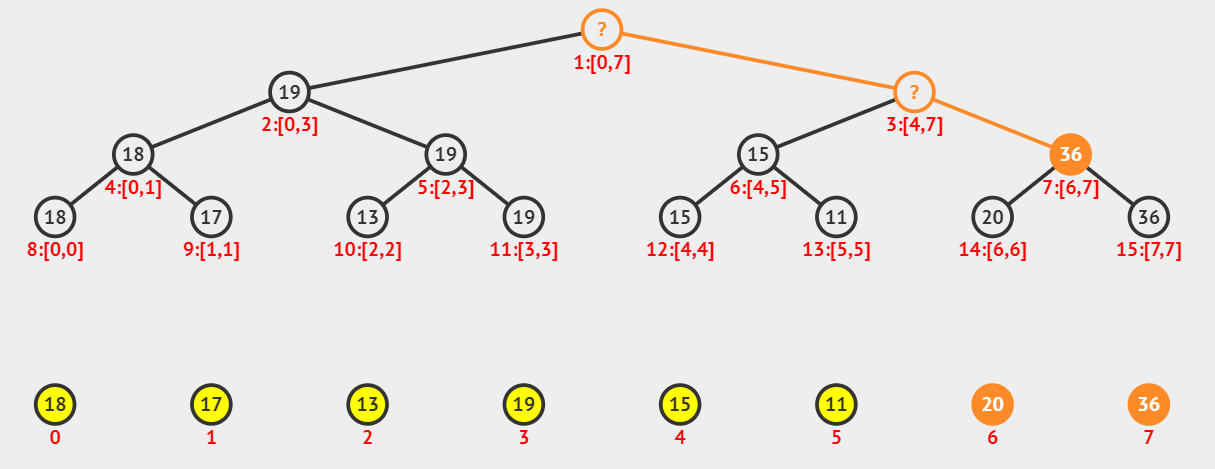


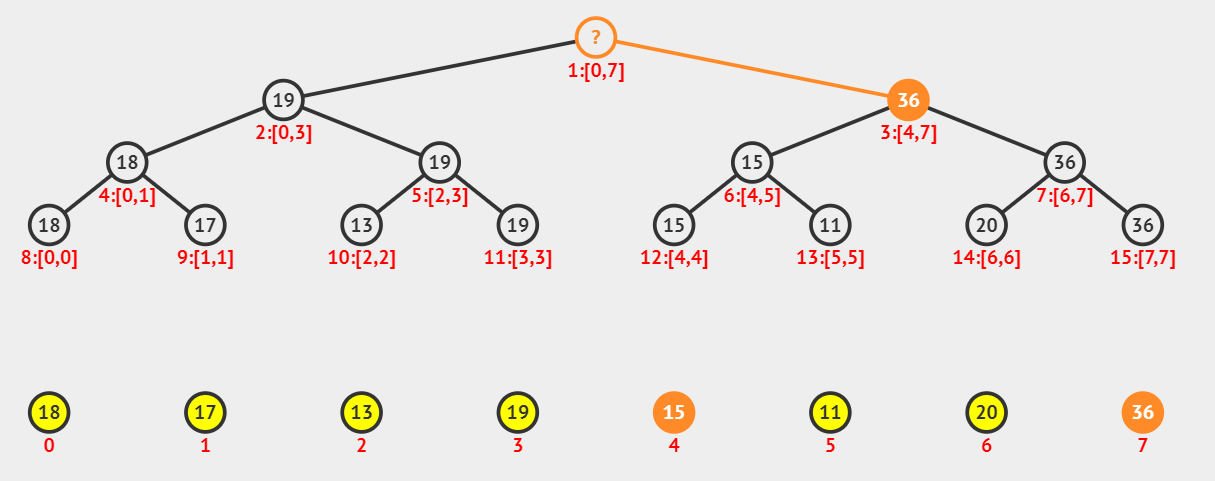


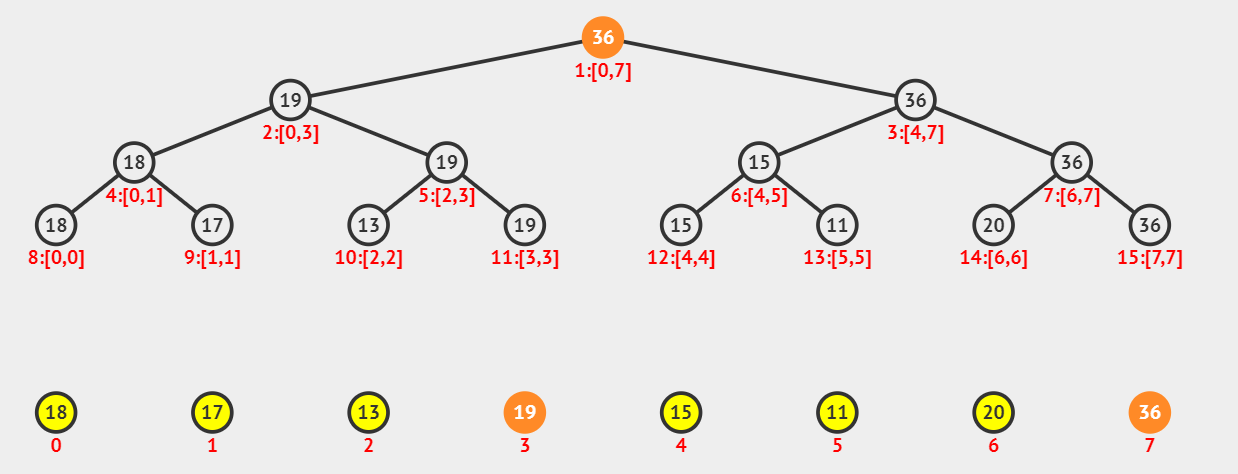












**Query operation:**

-Firstly, we start from the root.

-Secondly, check the current node a with the range is [l; r]:

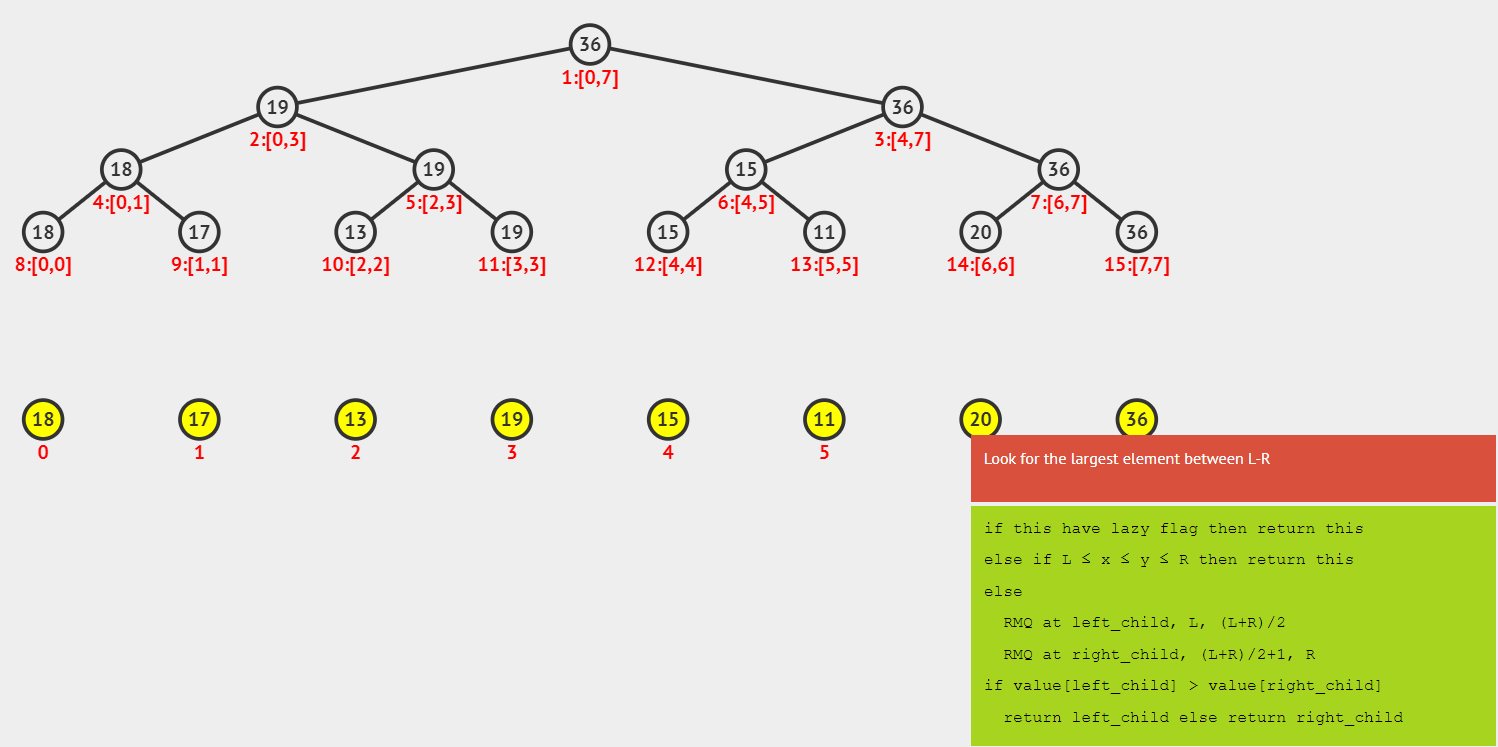
+)If [l,r] is in [i,j]: update the result then return

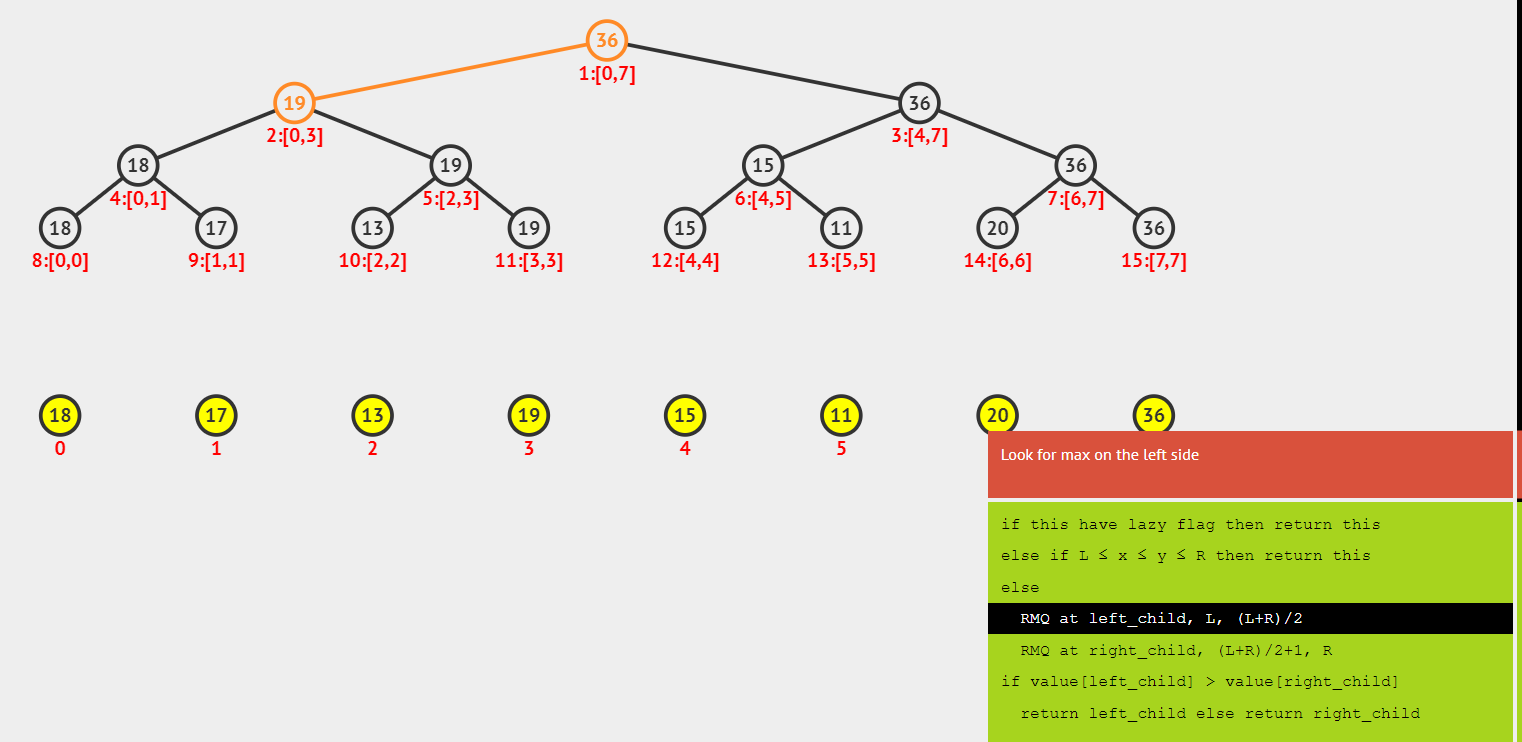
+)If [l,r] is out of [i,j]: return without update result

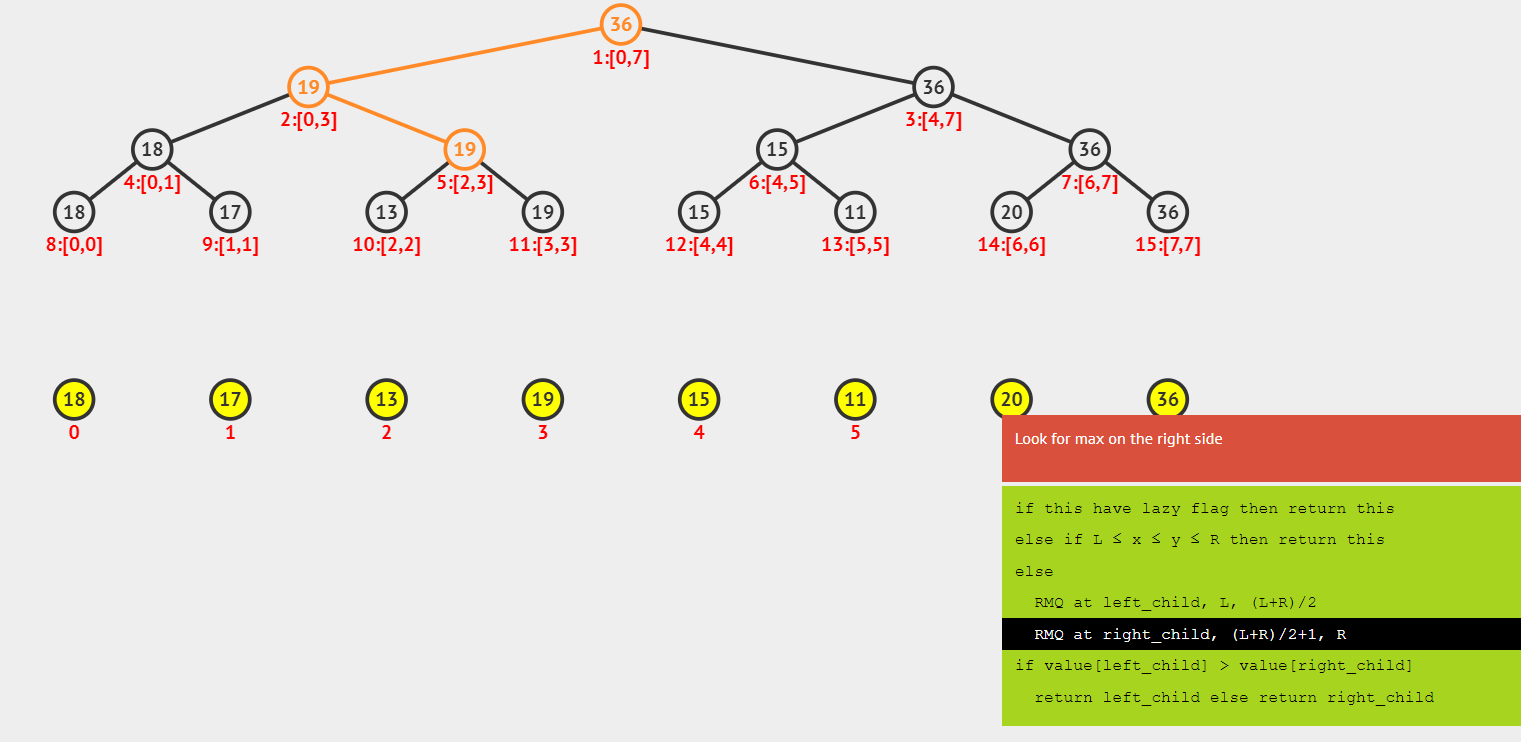
+)If [l,r] intersect [i,j]: recursively traverse to left and right child

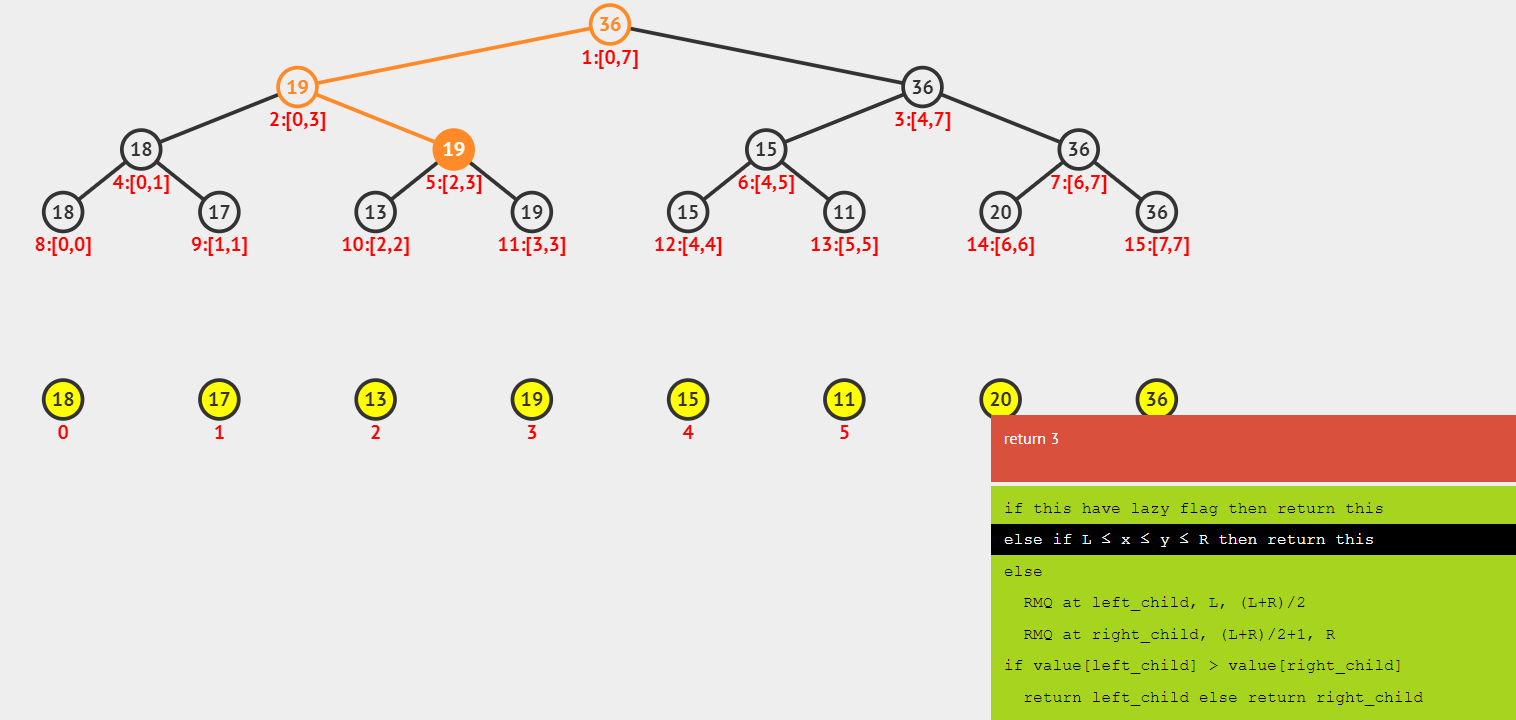
Complexity: O(log(n))

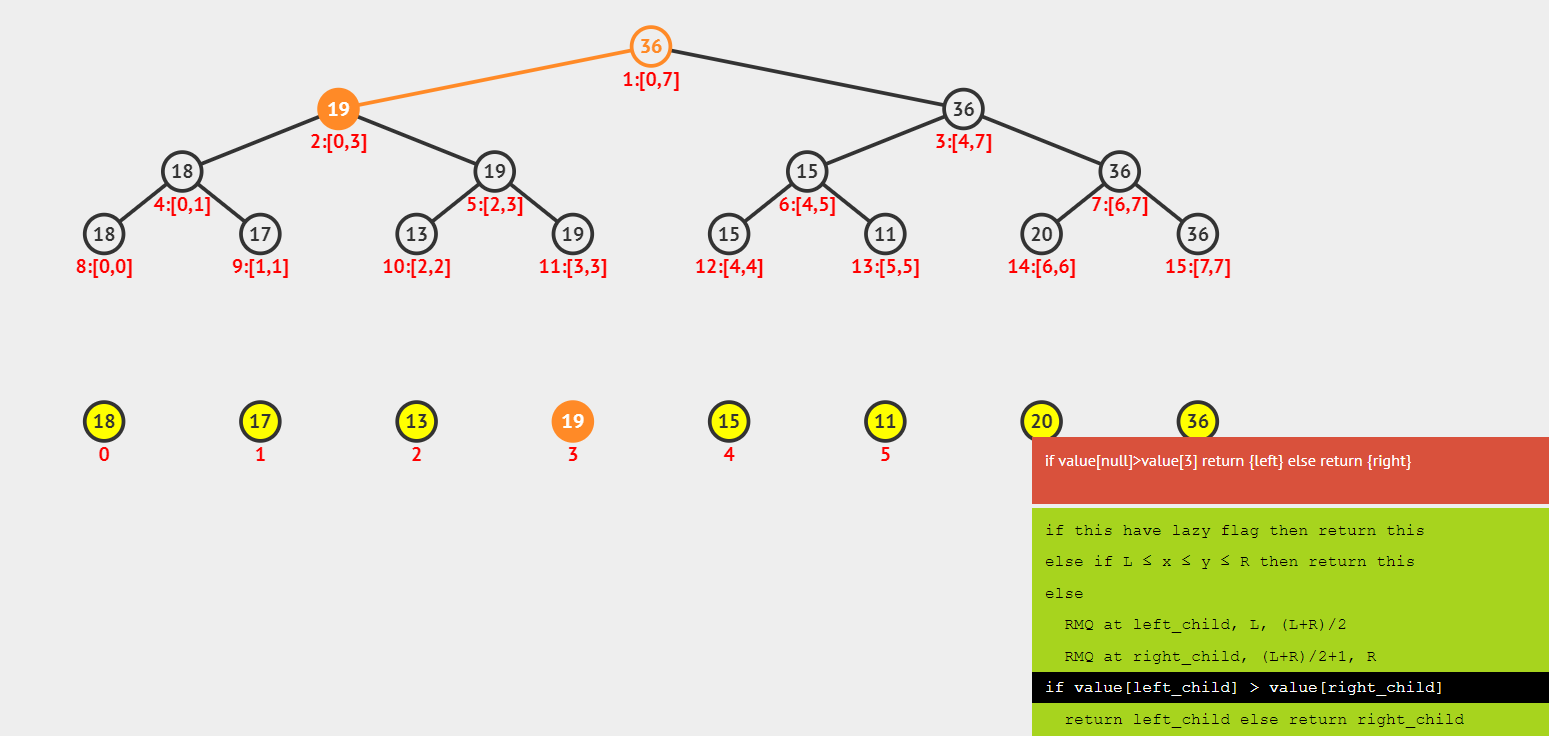
Example: Query[2; 5] with the segment tree that we built before:

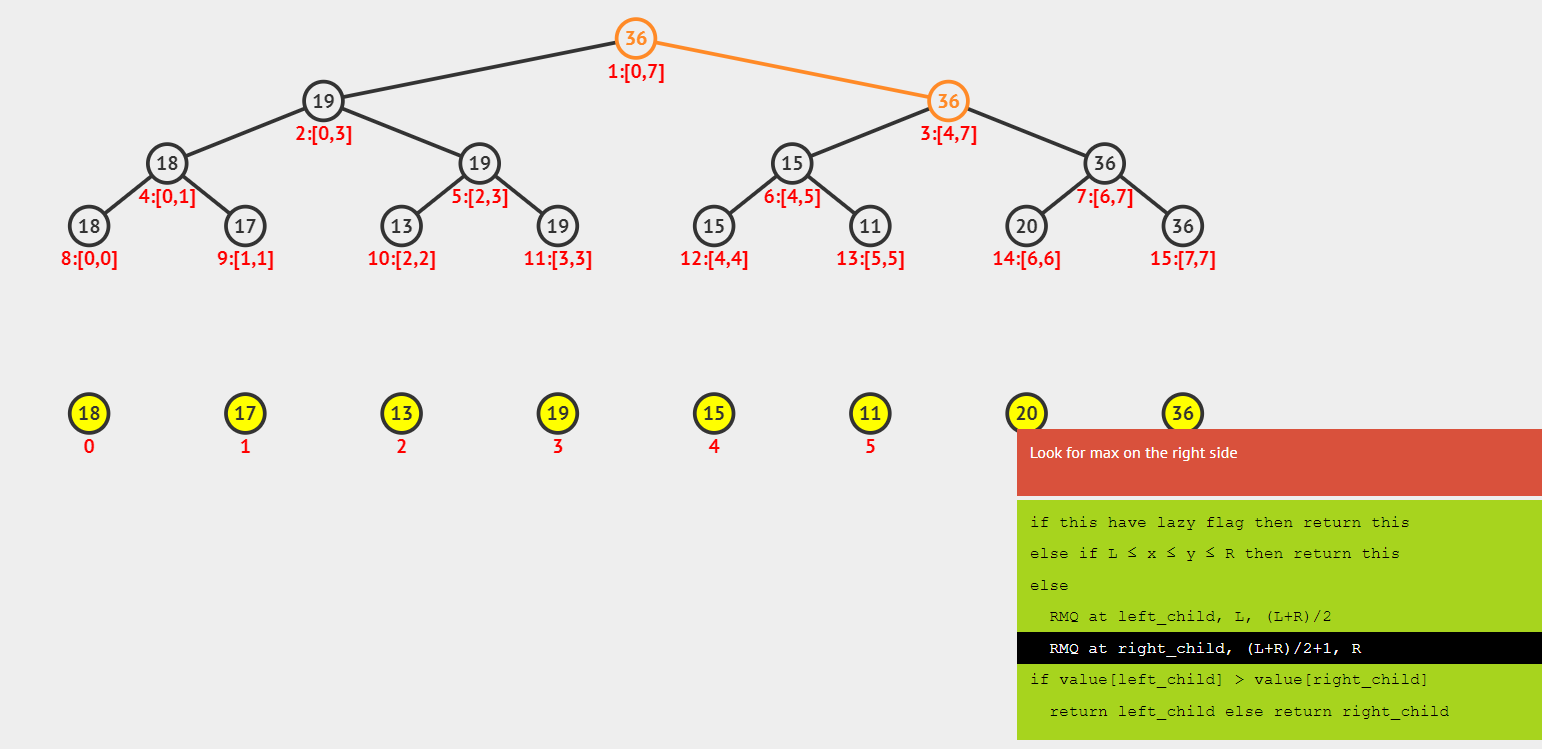


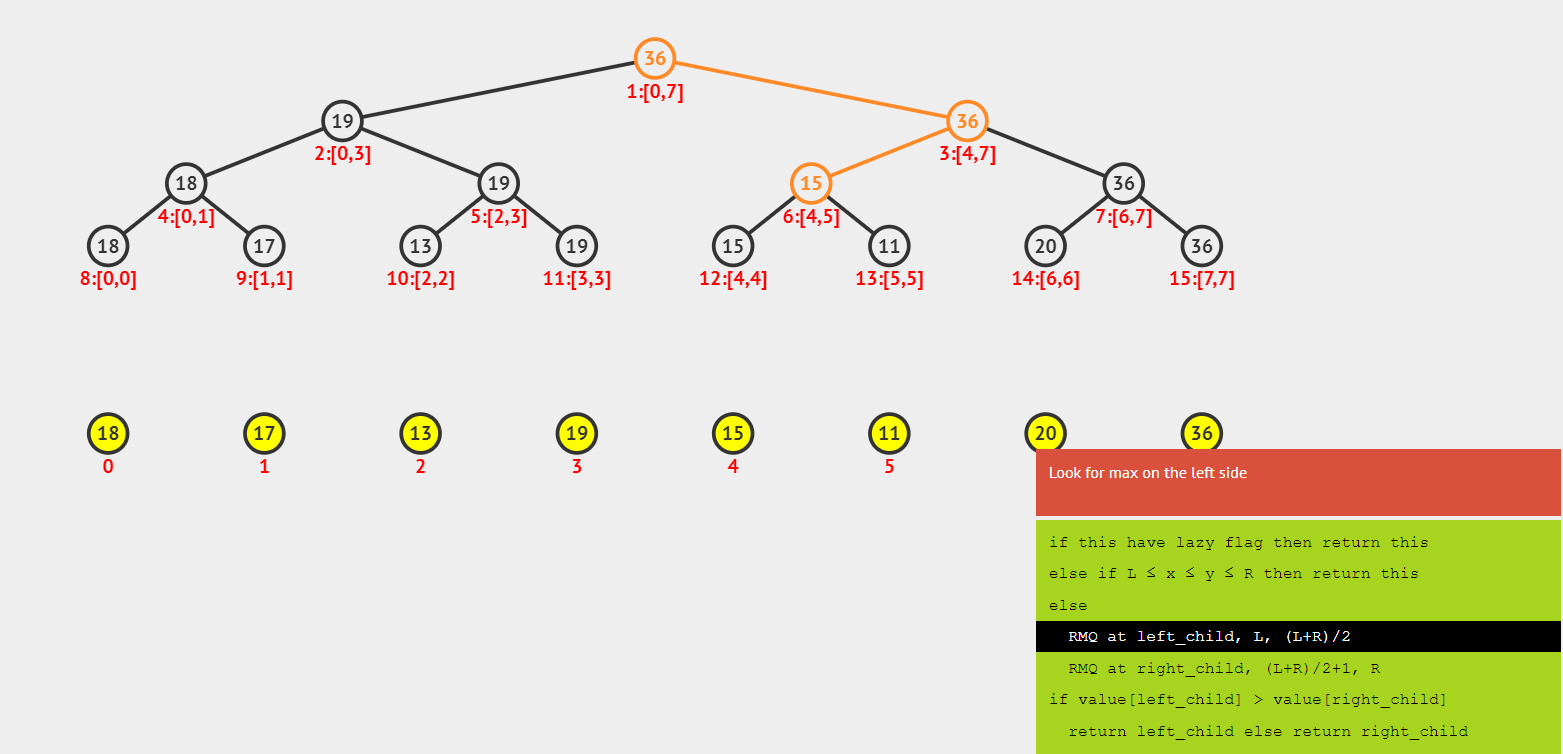


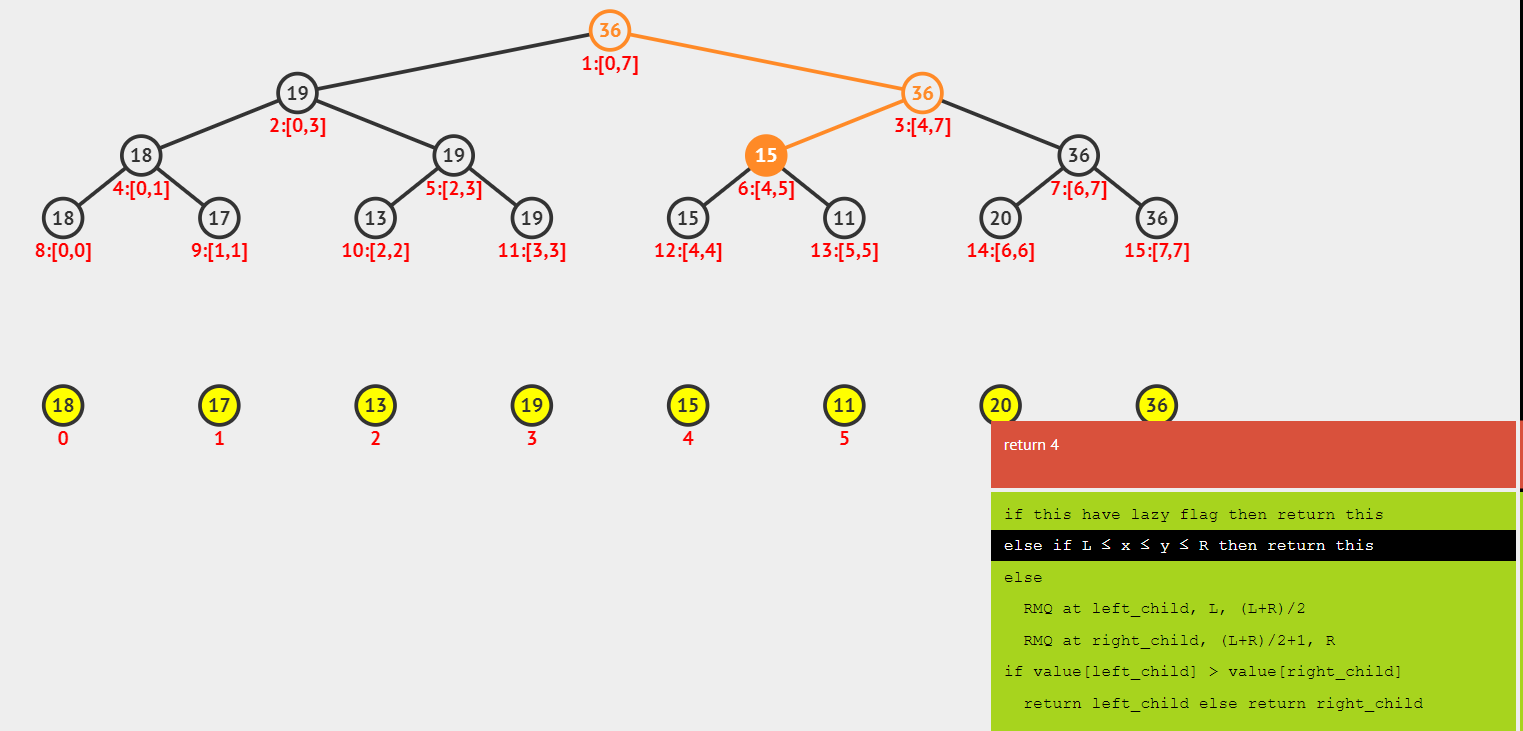


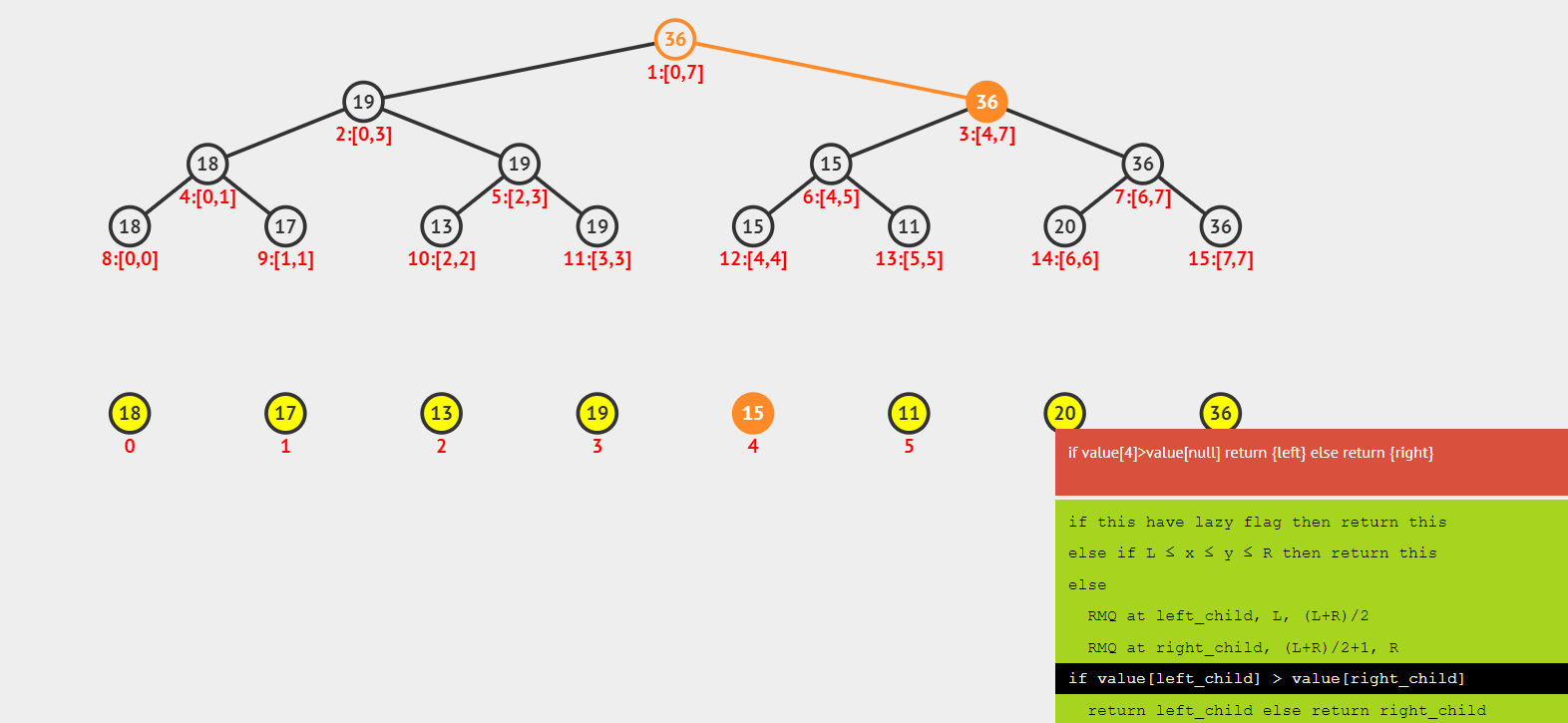


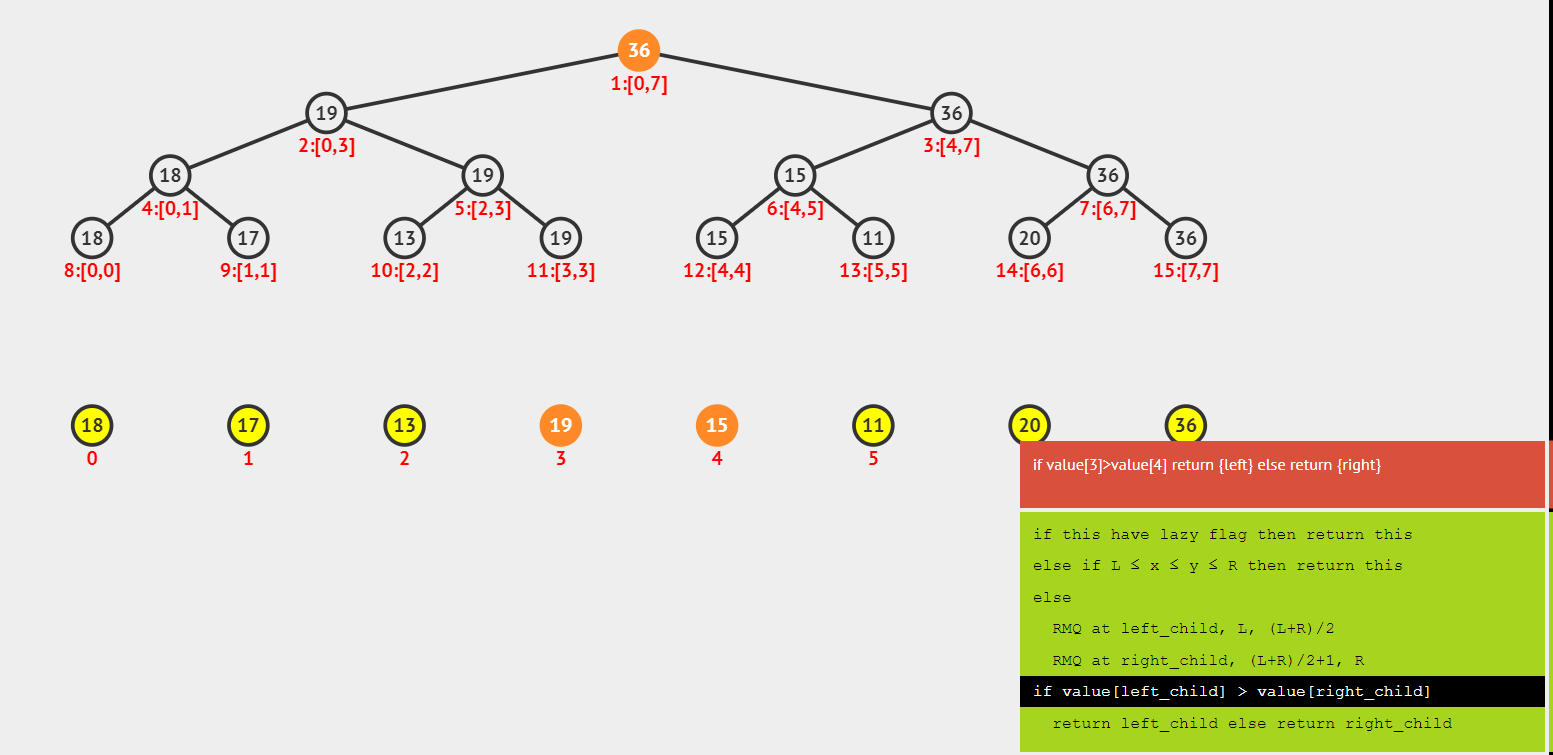


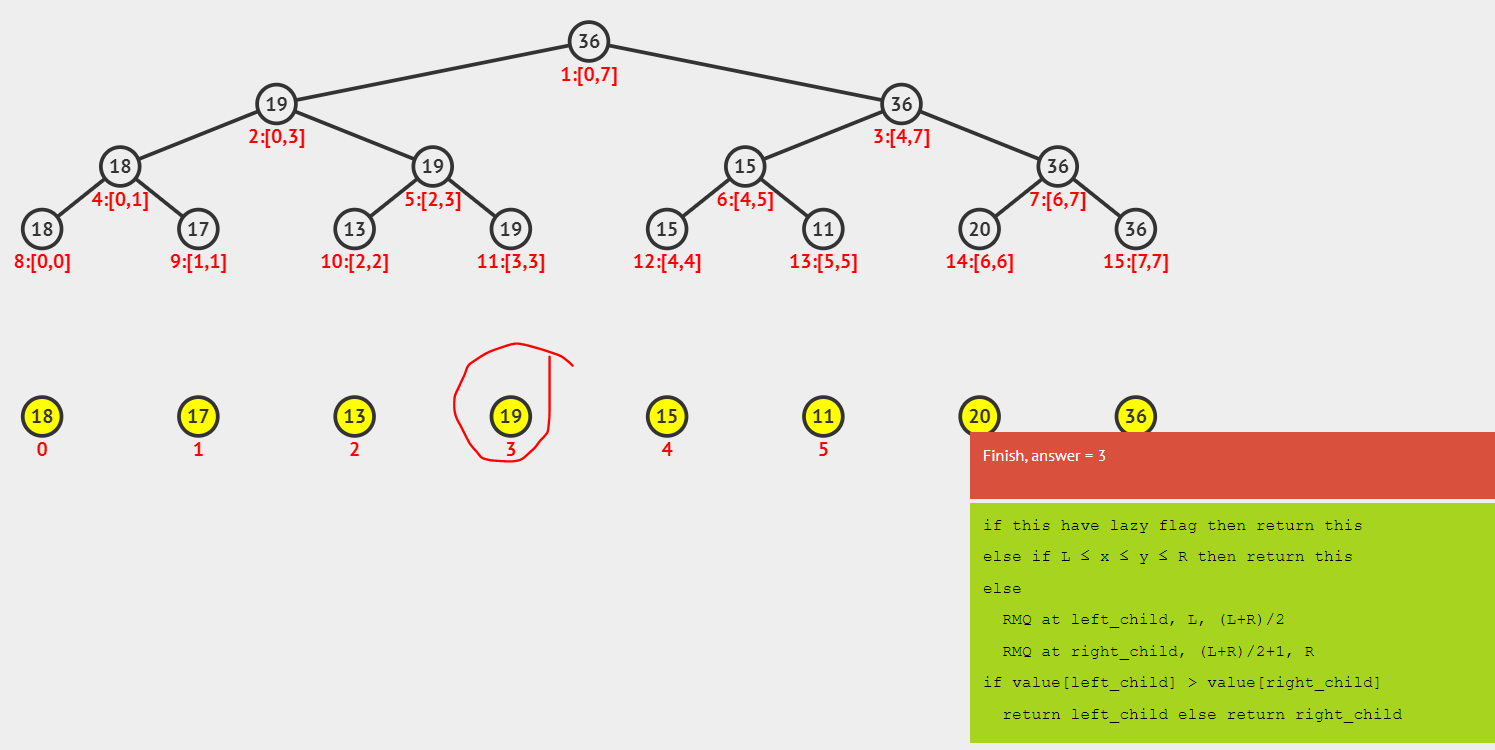












**Update Value operation** (1 element)

-Firstly, we assume the index of element updating is i.

-Start from root

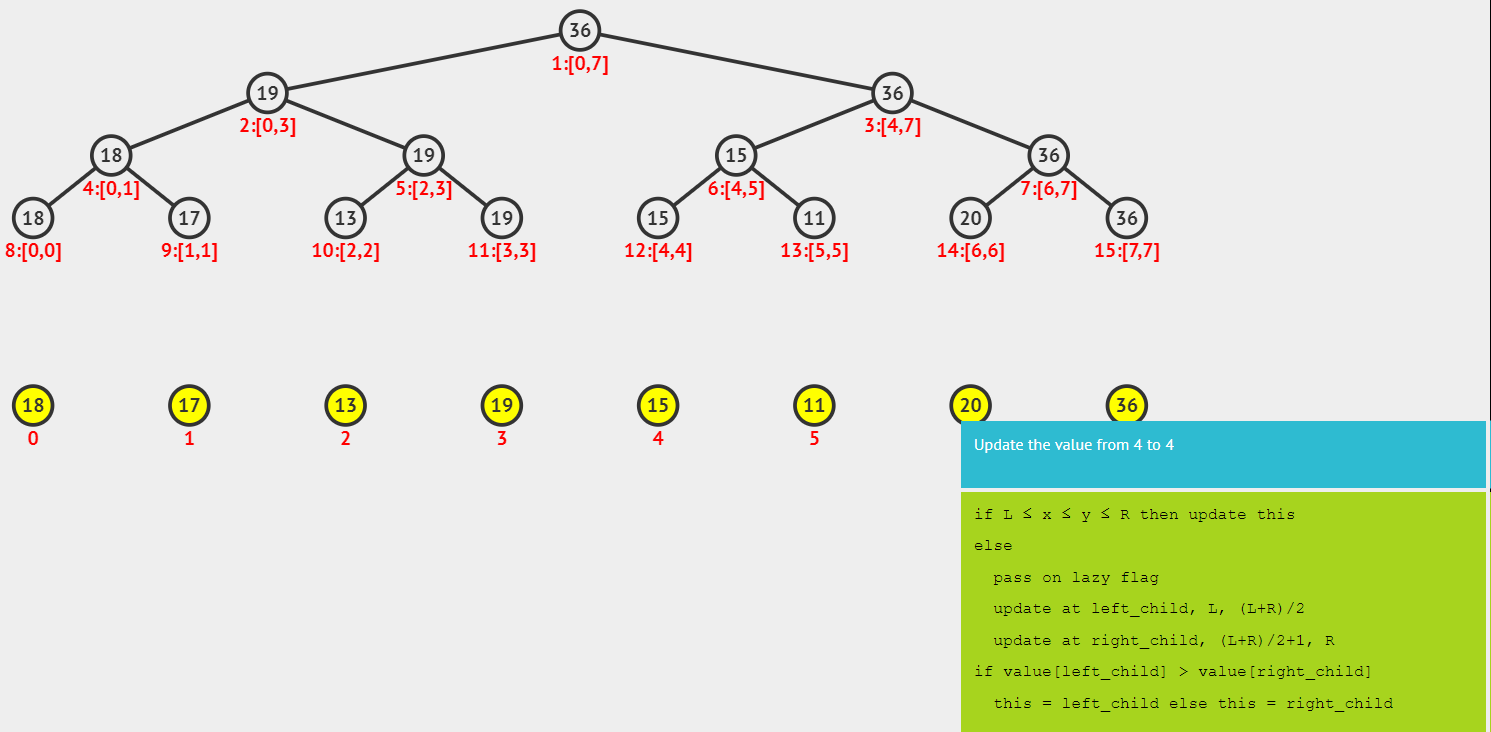
-Check the current node a which range is [l ;r]:

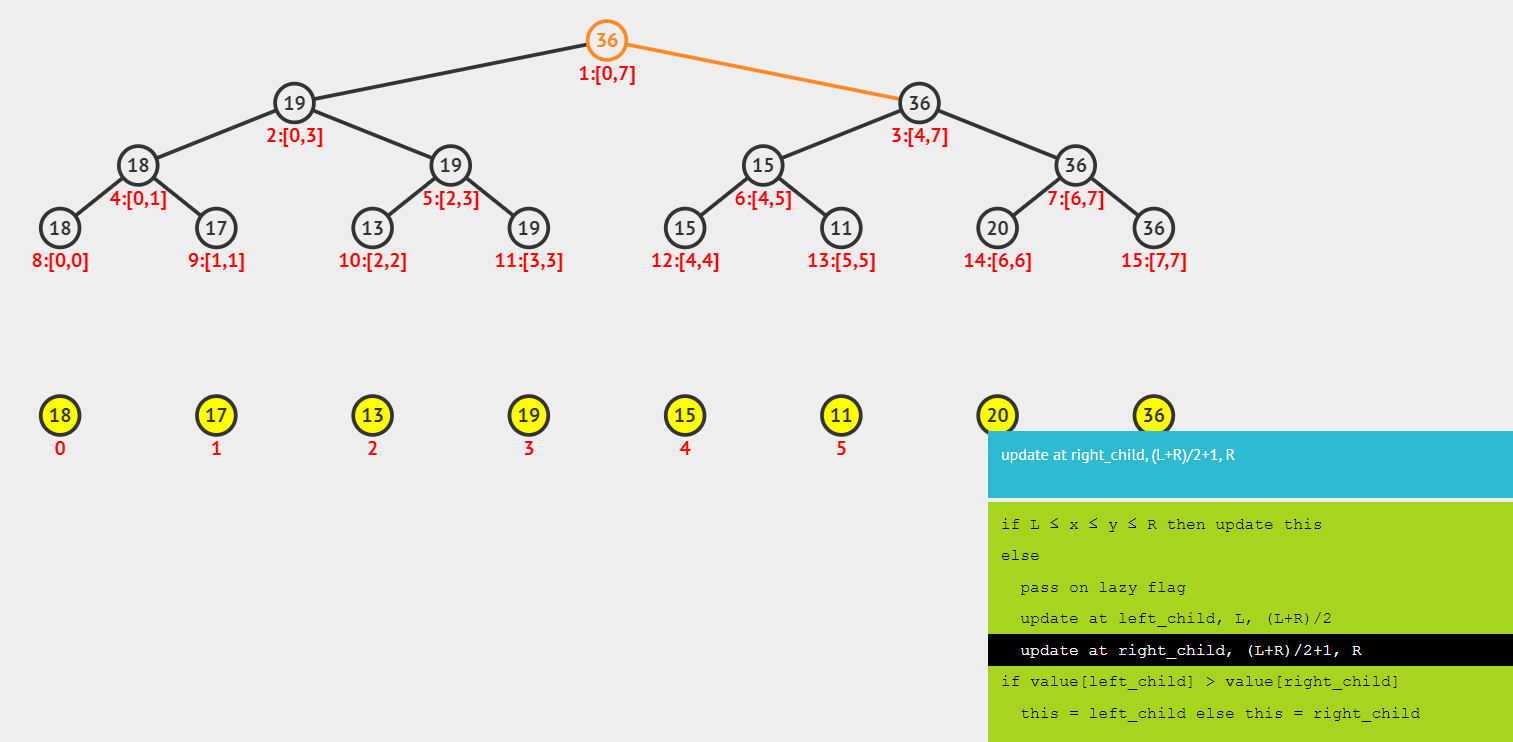
+)If i is not in [l,r]: return

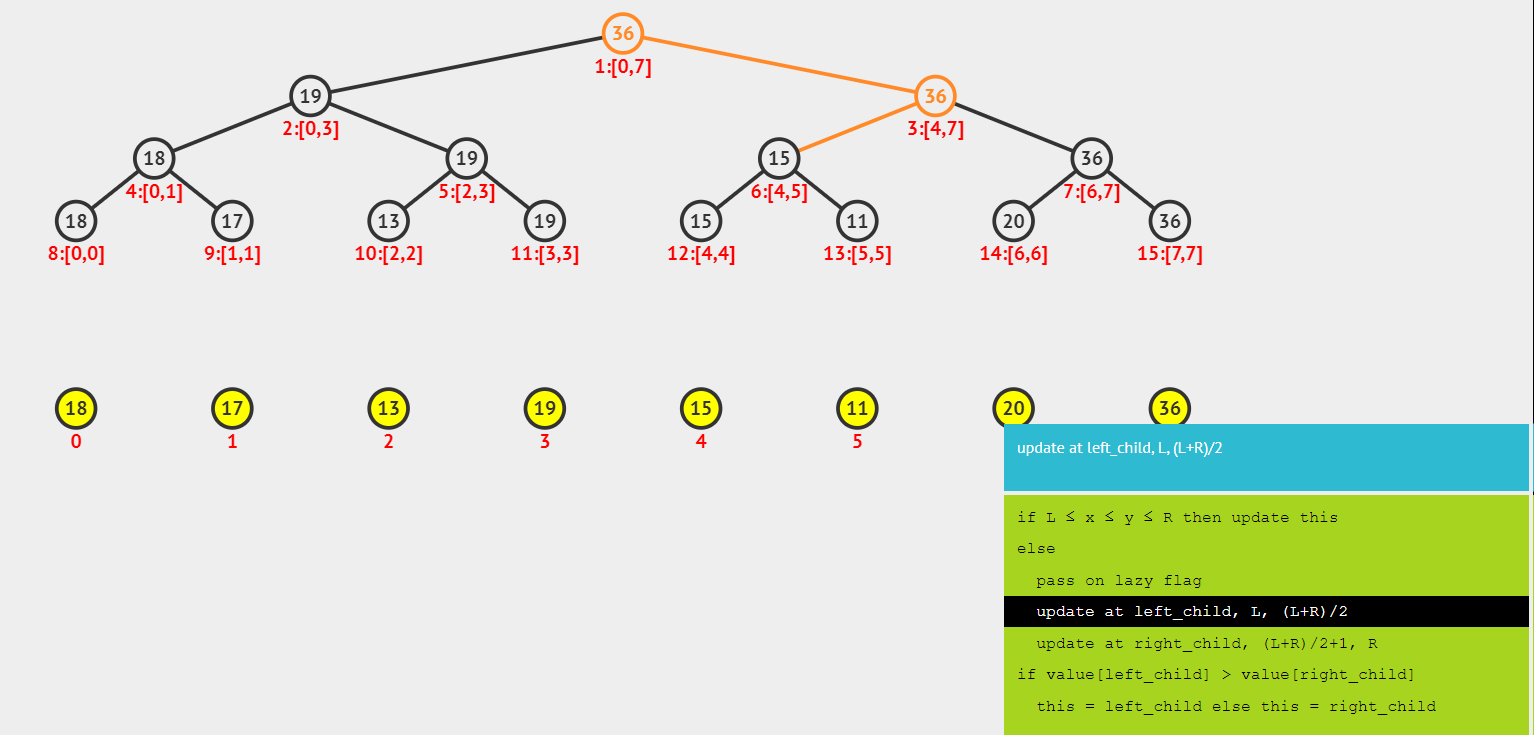
+)If l = r: update this node then return

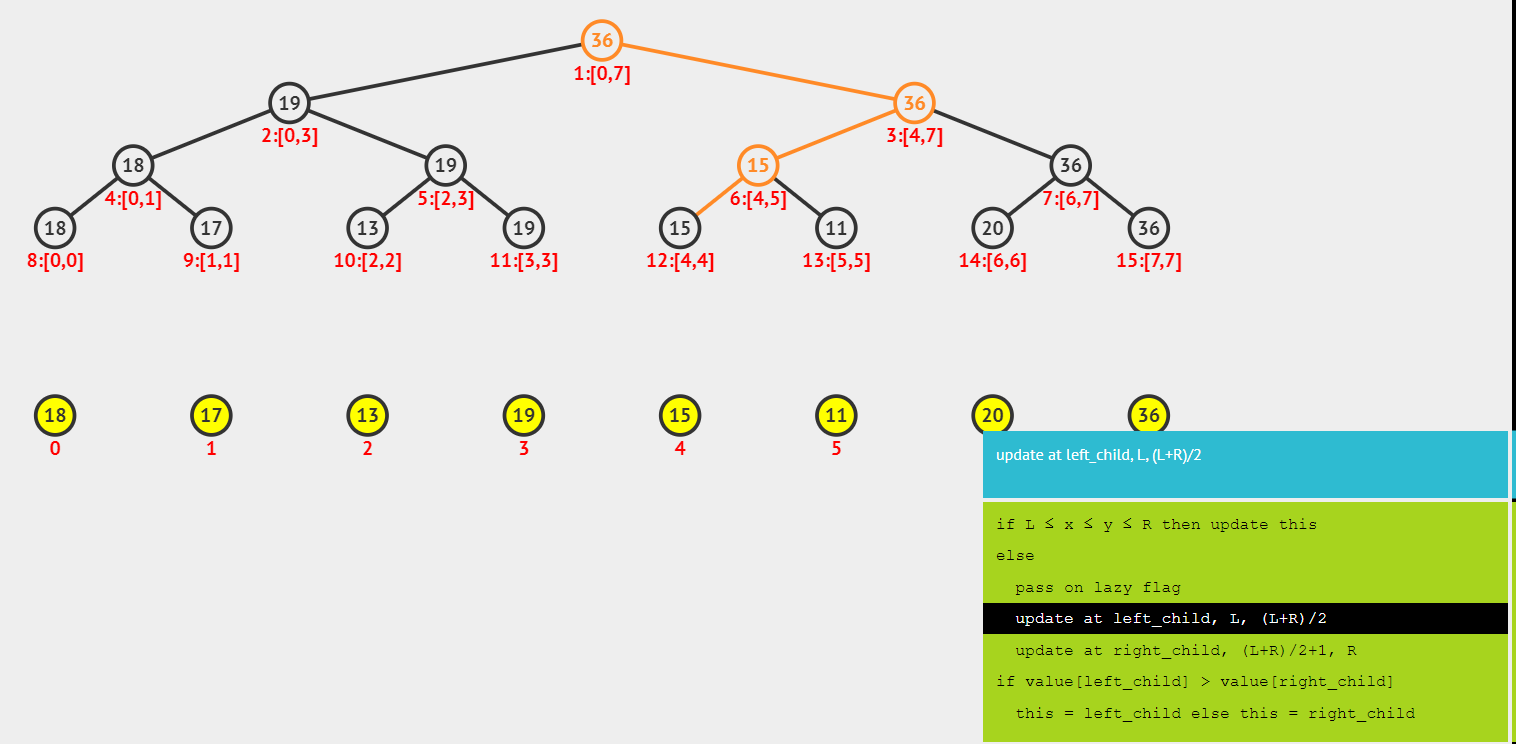
+)If i is in [l,r]: try to update left node and right node.

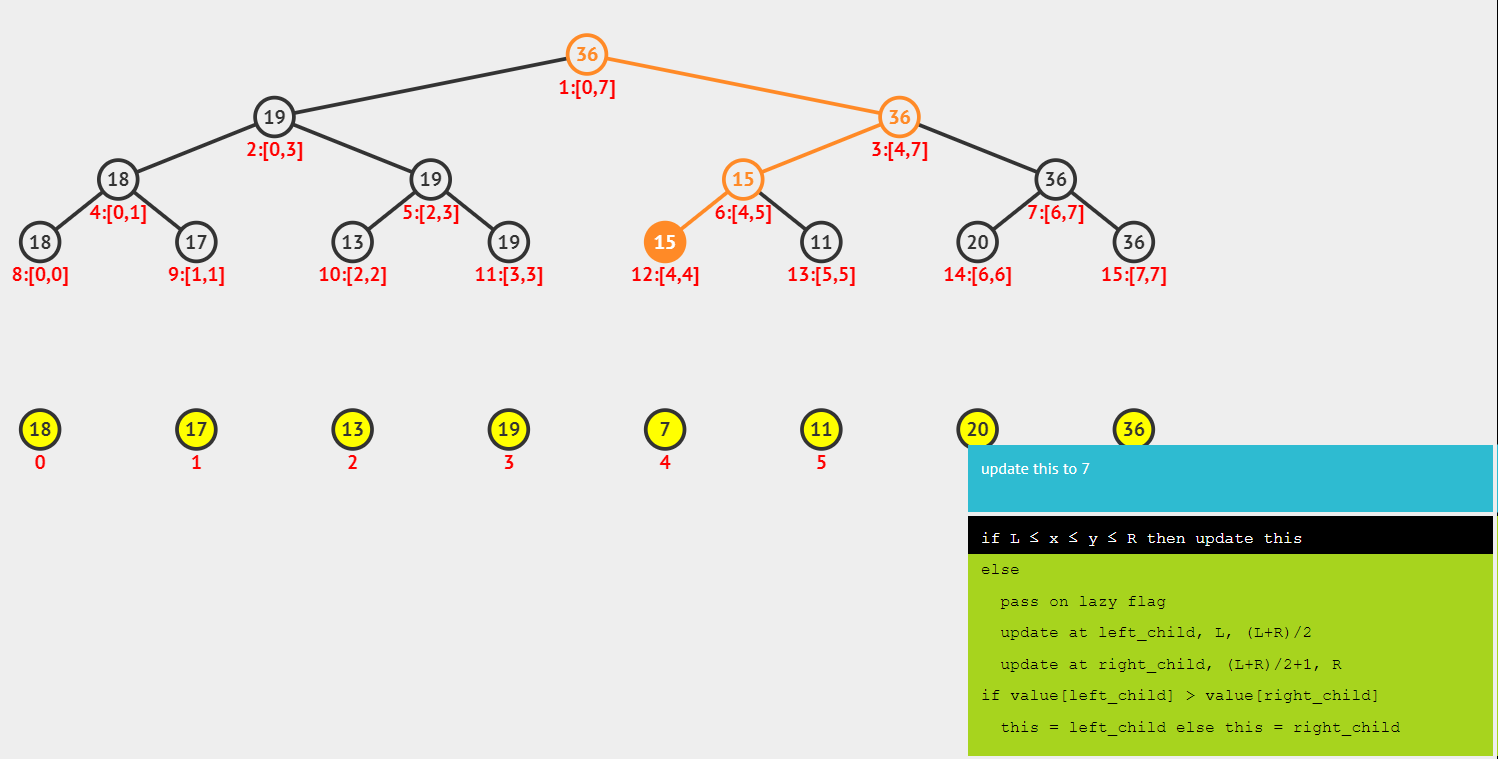
Example:

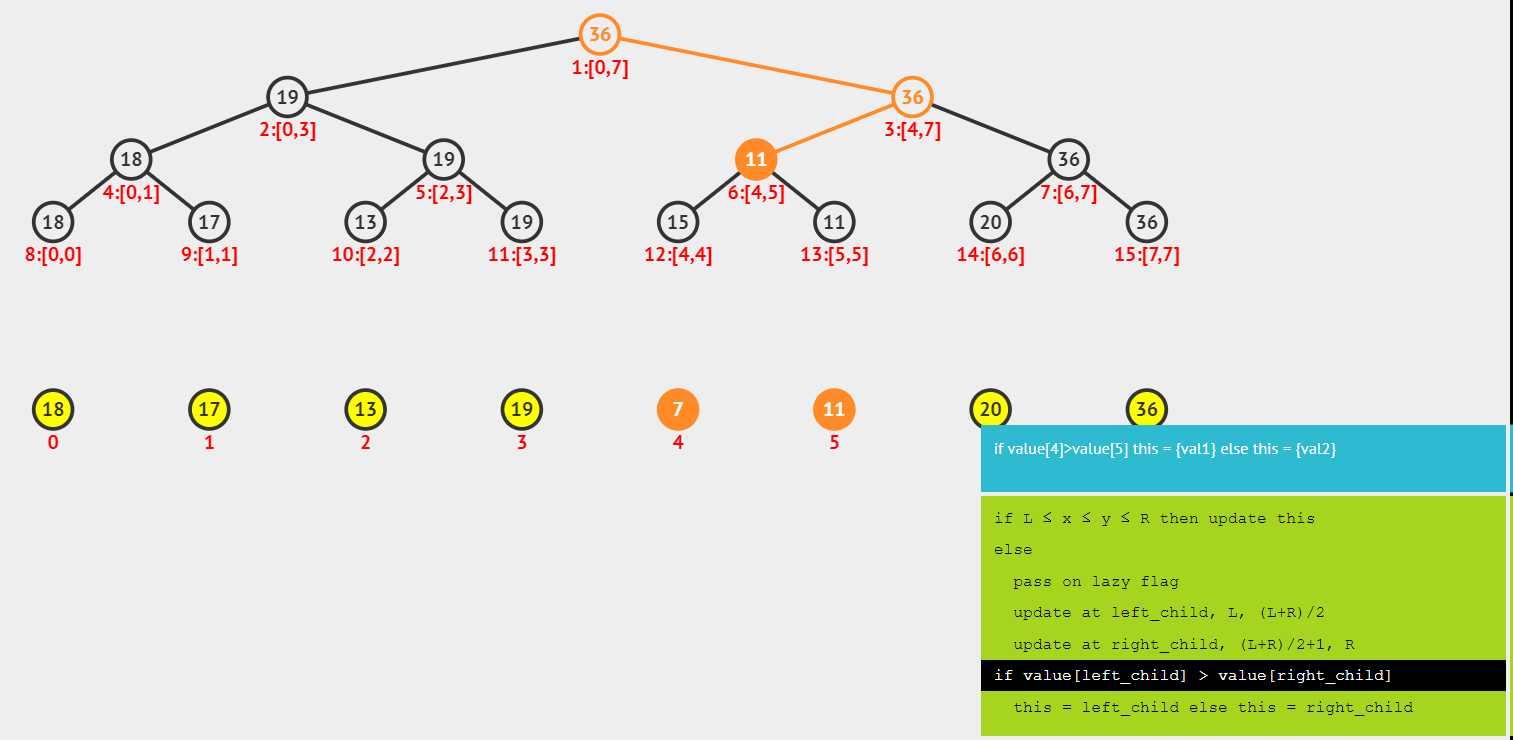


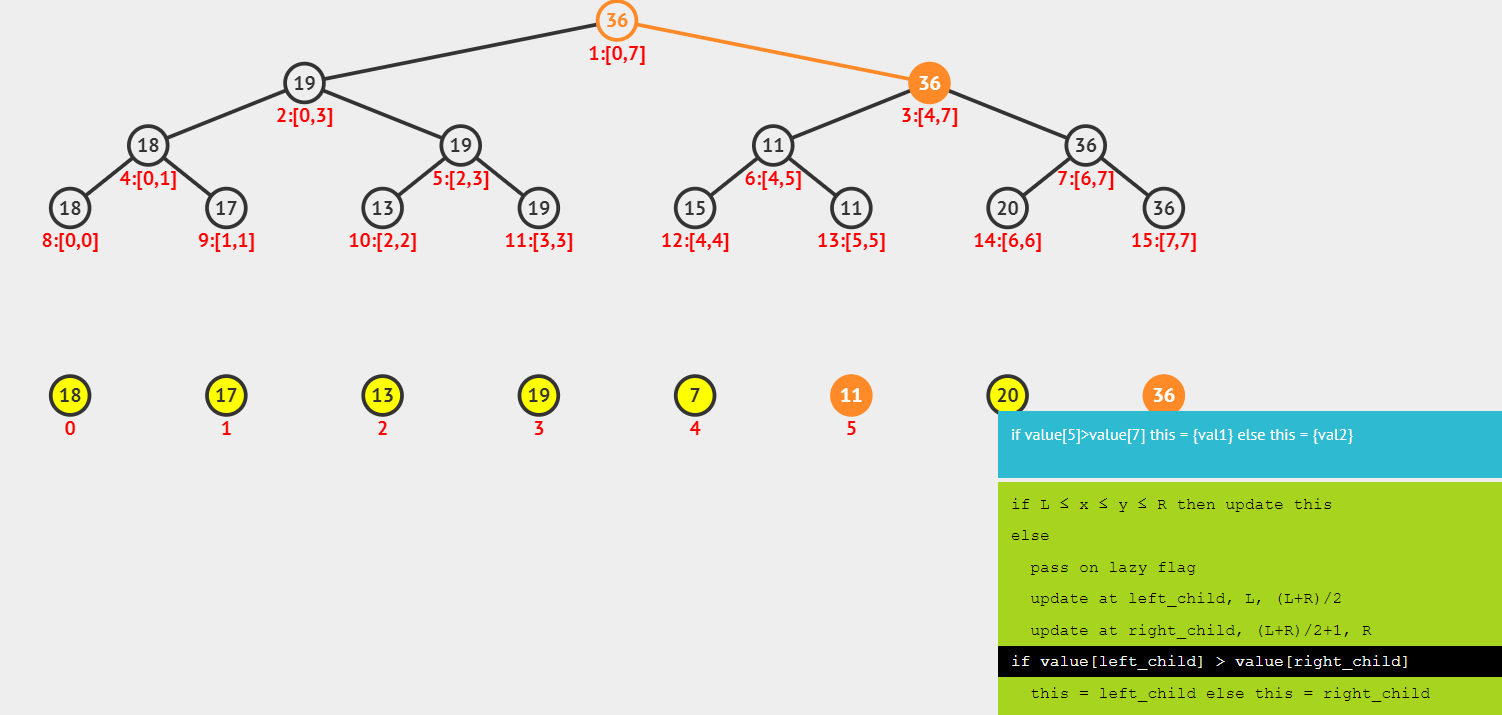


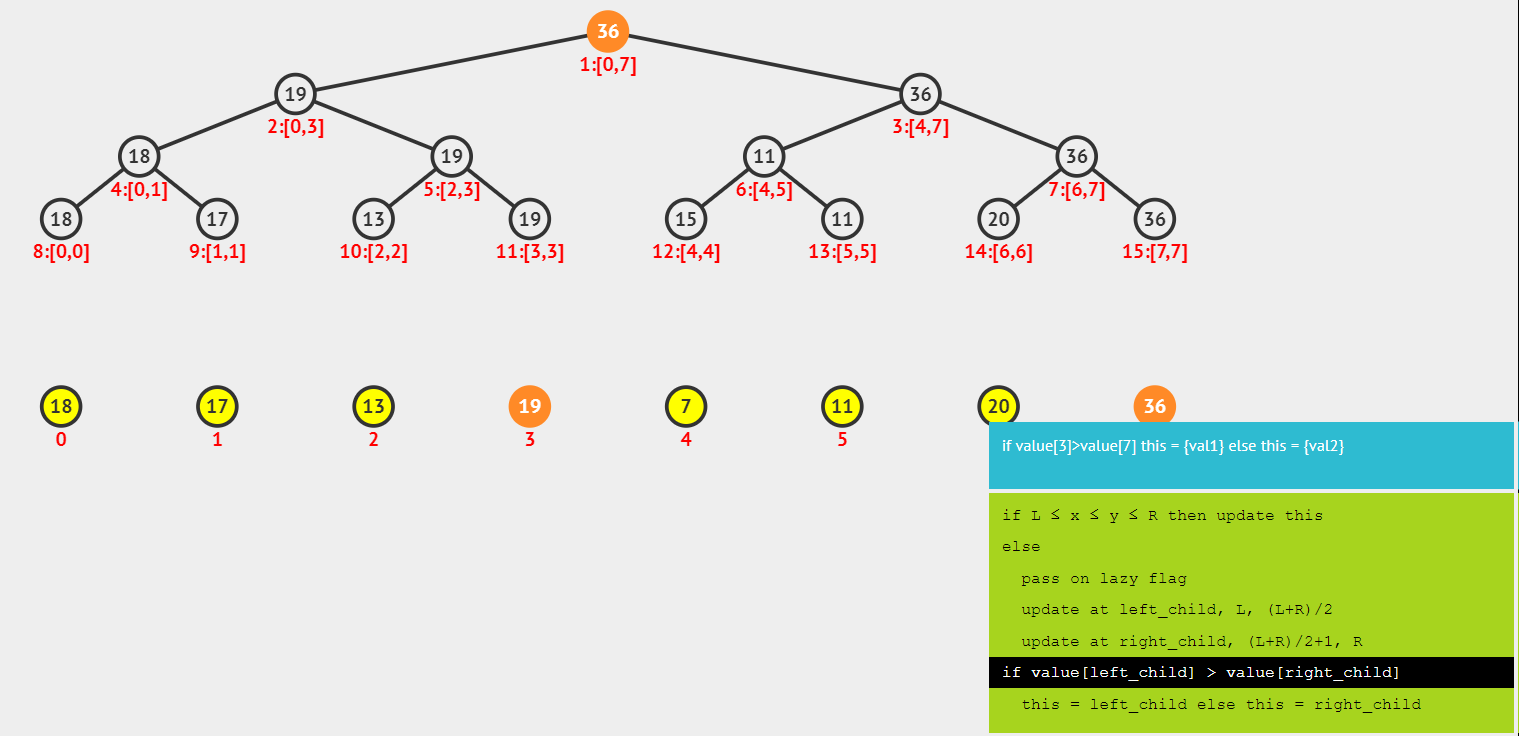


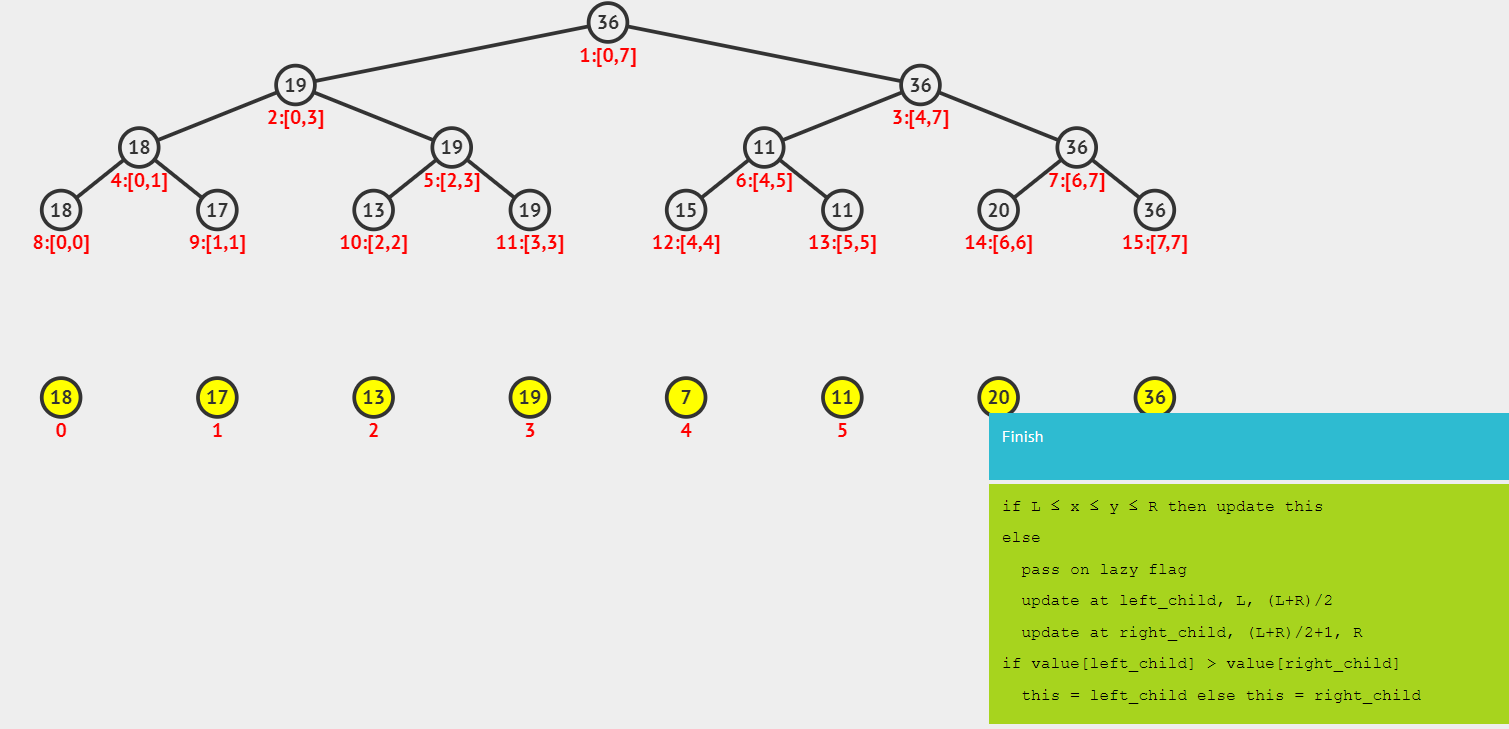












**3/Application:**

**a) Greatest common divisors of given ranges in an array:**

Given an array a[0 . . . n-1]. We should be able to efficiently find the GCD from index qs (query start) to qe (query end) where 0 <= qs <= qe <= n-1.

Example:

Input: a[] = {2, 3, 60, 90, 50};

Index Ranges: {1, 3}, {2, 4}, {0, 2}

Output: GCDs of given ranges are 3, 10, 1

Method solving: Segment Tree

**Representation of Segment trees**

* Leaf Nodes are the elements of the input array.
* Each internal node represents GCD of all leaves under it.

Array representation of tree is used to represent Segment Trees i.e., for each node at index i.

* Left child is at index 2\*i+1
* Right child at 2\*i+2 and the parent is at floor((i-1)/2).

**Construction of Segment Tree from given array**

* Begin with a segment arr[0 . . . n-1] and keep dividing into two halves. Every time we divide the current segment into two halves (if it has not yet become a segment of length 1), then call the same procedure on both halves, and for each such segment, we store the GCD value in a segment tree node.
* All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree (every node has 0 or two children) because we always divide segments in two halves at every level.
* Since the constructed tree is always full binary tree with n leaves, there will be n-1 internal nodes. So total number of nodes will be 2\*n – 1.
* Height of the segment tree will be &lceillog2n&rceil. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be 2\*2⌈log2n⌉ – 1

**Query for GCD of given range**

/ qs 🡪 query start index, qe 🡪 query end index

int GCD(node, qs, qe)

{

if range of node is within qs and qe

return value in node

else if range of node is completely

outside qs and qe

return INFINITE

else

return GCD( GCD(node's left child, qs, qe),

GCD(node's right child, qs, qe) )

}

**Implementation:**

#include <iostream>

using namespace std;

// To store segment tree

int \*st;

/\*  A recursive function to get gcd of given

    range of array indexes. The following are parameters for

    this function.

    st    --> Pointer to segment tree

    si --> Index of current node in the segment tree. Initially

               0 is passed as root is always at index 0

    ss & se  --> Starting and ending indexes of the segment

                 represented by current node, i.e., st[index]

    qs & qe  --> Starting and ending indexes of query range \*/

int findGcd(int ss, int se, int qs, int qe, int si)

{

    if (ss>qe || se < qs)

        return 0;

    if (qs<=ss && qe>=se)

        return st[si];

    int mid = ss+(se-ss)/2;

    return \_\_gcd(findGcd(ss, mid, qs, qe, si\*2+1),

               findGcd(mid+1, se, qs, qe, si\*2+2));

}

//Finding The gcd of given Range

int findRangeGcd(int ss, int se, int arr[],int n)

{

    if (ss<0 || se > n-1 || ss>se)

    {

        cout << "Invalid Arguments" << "\n";

        return -1;

    }

    return findGcd(0, n-1, ss, se, 0);

}

// A recursive function that constructs Segment Tree for

// array[ss..se]. si is index of current node in segment

// tree st

int constructST(int arr[], int ss, int se, int si)

{

    if (ss==se)

    {

        st[si] = arr[ss];

        return st[si];

    }

    int mid = ss+(se-ss)/2;

    st[si] = \_\_gcd(constructST(arr, ss, mid, si\*2+1),

                 constructST(arr, mid+1, se, si\*2+2));

    return st[si];

}

/\* Function to construct segment tree from given array.

   This function allocates memory for segment tree and

   calls constructSTUtil() to fill the allocated memory \*/

int \*constructSegmentTree(int arr[], int n)

{

   int height = (int)(ceil(log2(n)));

   int size = 2\*(int)pow(2, height)-1;

   st = new int[size];

   constructST(arr, 0, n-1, 0);

   return st;

}

// Driver program to test above functions

int main()

{

    int a[] = {2, 3, 6,12 , 5};

    int n = sizeof(a)/sizeof(a[0]);

    // Build segment tree from given array

    constructSegmentTree(a, n);

    // Starting index of range. These indexes are 0 based.

    int l = 1;

    // Last index of range.These indexes are 0 based.

    int r = 3;

    cout << "GCD of the given range is:";

    cout << findRangeGcd(l, r, a, n) << "\n";

    return 0;

}

Output: 3

Time Complexity for tree construction is O(n \* log(min(a, b))), where n is the number of modes and a and b are nodes whose GCD is calculated during merge operation. There are total 2n-1 nodes, and value of every node is calculated only once in tree construction. Time complexity to query is O(Log n \* Log n).

**b)Find the k-largest element in an array:**

Approach:

+First, we create a set S as an empty set

We have 2 queries:

+Query 1: Insert a value into set S

+Query 2: Finding the k-largest element in S.

Note: All values that we add in the query step is on the interval [1; maxValue]

Example:

 S = {}  
+ Insert 1 -> S = {1}  
+ Insert 2 -> S = {1, 2}  
+ Find 1 -> return 1  
+ Insert 1 -> S = {1, 1, 2}  
+ Find 2 -> return 1  
+ Find 3 -> return 2

-Each leaf nodes in the segment tree now are showing the value of element in set S. It is meaning that the node with position i contains the number of value equaling i in the set S.

-To find the k-largest element in the set S, we have a lemma:

When we go down from the root, the sub-left nodes contain information relative to the number of value in S that all of these values are on the interval [X; Y]. The right – sub nodes also contains the information about the number of value in S that all of them are on the interval [Y +1; Z]. Suppose that the sub-left node contain element. If P K then the K –The Largest element will locate at the left-sub tree that we are considering. In contrast, it will locate at the right – sub tree. Next, we will find the (K – P) largest numbers in the sub-tree(Because we have found that there are P values smaller than the K – largest number that locate on the left –sub tree).

Build: No

Update: O(log(maxValue))

Query: O(log(maxValue))

**c) Find k-smallest elements in a subarray:**

Given an array **arr** of size **N**. The task is to find the kth smallest element in the subarray(l to r, both inclusive).

**Note :**

* Query are of type query(l, r, k)
* 1 <= k <= r-l+1
* There can be multiple queries.

**Examples:**

***Input :****arr = {3, 2, 5, 4, 7, 1, 9}, query = (2, 6, 3)****Output :****4   
sorted subarray in a range 2 to 6 is {1, 2, 4, 5, 7} and 3rd element is 4*

***Input :****arr = {2, 3, 4, 1, 6, 5, 8}, query = (1, 5, 2)****Output :****2*

Let, S = r - l + 1.

**Naive Approach:**

* Copy the subarray into some other local array. After sorting find the kth element.   
  **Time complexity:** Slog(S)
* Use a max priority queue ‘p’ and iterate in the subarray. If size of ‘p’ is less than ‘k’ insert element else remove top element and insert the new element into ‘p’ after complete interaction top of ‘p’ will be the answer.  
  **Time complexity:** Slog(k)

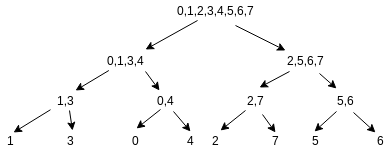
**Efficient Approach:** The idea is to use [Segment trees](https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/), to be more precise use [merge sort segment tree](https://www.geeksforgeeks.org/merge-sort-tree-smaller-or-equal-elements-in-given-row-range/). Here, Instead of storing sorted elements we store indexes of sorted elements.

Let **B** is the array after sorting **arr** and **seg** is our segment tree. Node **ci** of **seg** stores the sorted order of indices of **arr** which are in range **[st, end]**.

**If arr = {3, 1, 5, 2, 4, 7, 8, 6},**

**then B is {1, 2, 3, 4, 5, 6, 7, 8}**

Segment tree will look like :



Let’s suppose seg[ci]->left holds **p** elements. If **p** is less then or equals to **k**, we can find kth smallest in left child and if **p** is less than **k** then move to right child and find **(k-p)** smallest element.

One can find the number of elements in the sorted array(A) lying in between elements X and Y by:

*upper\_bound(A.begin(), A.end(), Y)-lower\_bound(A.begin(), A.end(), X)*

**Time complexity:**  
To build segment tree: O(n\*log(n))   
For each query : O(log(n)\*log(n))

Implementation:

// C++ program to find the kth smallest element in a range

#include <bits/stdc++.h>

using namespace std;

#define N (int)1e5

// Declaring a global segment tree

vector<int> seg[N];

// Function to build the merge sort

// segment tree of indices

void build(int ci, int st, int end,

           pair<int, int>\* B)

{

    if (st == end) {

        // Using second property of B

        seg[ci].push\_back(B[st].second);

        return;

    }

    int mid = (st + end) / 2;

    build(2 \* ci + 1, st, mid, B);

    build(2 \* ci + 2, mid + 1, end, B);

    // Inbuilt merge function

    // this takes two sorted arrays and merge

    // them into a sorted array

    merge(seg[2 \* ci + 1].begin(), seg[2 \* ci + 1].end(),

          seg[2 \* ci + 2].begin(), seg[2 \* ci + 2].end(),

          back\_inserter(seg[ci]));

}

// Function to return the index of

// kth smallest element in range [l, r]

int query(int ci, int st, int end,

          int l, int r, int k)

{

    // Base case

    if (st == end)

        return seg[ci][0];

    // Finding value of 'p' as described in article

    // seg[2\*ci+1] is left node of seg[ci]

    int p = upper\_bound(seg[2 \* ci + 1].begin(),

                        seg[2 \* ci + 1].end(), r)

            - lower\_bound(seg[2 \* ci + 1].begin(),

                          seg[2 \* ci + 1].end(), l);

    int mid = (st + end) / 2;

    if (p >= k)

        return query(2 \* ci + 1, st, mid, l, r, k);

    else

        return query(2 \* ci + 2, mid + 1, end, l, r, k - p);

}

// Driver code

int main()

{

    int arr[] = { 3, 1, 5, 2, 4, 7, 8, 6 };

    int n = sizeof(arr) / sizeof(arr[0]);

    pair<int, int> B[n];

    for (int i = 0; i < n; i++) {

        B[i] = { arr[i], i };

    }

    // After sorting, B's second property is

    // something upon which we will build our Tree

    sort(B, B + n);

    // Build the tree

    build(0, 0, n - 1, B);

    cout << "3rd smallest element in range 3 to 7 is: "

         << arr[query(0, 0, n - 1, 2, 6, 3)] << "\n";

}

Output: 3rd smallest element in range 3 to 7 is: 5

**4/Technological innovation of Segment Tree:**

**a) Lazy propagation (Updating an Interval):**

Sometimes problems will ask you to update an interval from **l to r**, instead of a single element. One solution is to update all the elements one by one. Complexity of this approach will be **O(N)** per operation since where are **N** elements in the array and updating a single element will take **O(logN)** time.

Let's be **Lazy** i.e., do work only when needed. How ? When we need to update an interval, we will update a node and mark its child that it needs to be updated and update it when needed. For this we need an array **lazy[]** of the same size as that of segment tree. Initially all the elements of the **lazy[]** array will be **0** representing that there is no pending update. If there is non-zero element **lazy[k]** then this element needs to update node **k** in the segment tree before making any query operation.

To update an interval we will keep 3 things in mind.

1. If current segment tree node has any pending update, then first add that pending update to current node.
2. If the interval represented by current node lies completely in the interval to update, then update the current node and update the **lazy[]** array for children nodes.
3. If the interval represented by current node overlaps with the interval to update, then update the nodes as the earlier update function

Since we have changed the update function to postpone the update operation, we will have to change the query function also. The only change we need to make is to check if there is any pending update operation on that node. If there is a pending update operation, first update the node and then work same as the earlier query function.

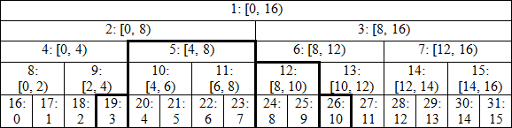
**b)Innovate space and run – time of segment tree:**

[**https://vnoi.info/wiki/algo/data-structures/segment-tree-extend.md**](https://vnoi.info/wiki/algo/data-structures/segment-tree-extend.md)

**https://codeforces.com/blog/entry/18051**

-Space: O(2n)

-Complexity: O(log(n))



**c)Persistent segment Tree:**

Segment Tree is itself a great data structure that comes into play in many cases. In this post we will introduce the concept of Persistency in this data structure. Persistency, simply means to retain the changes. But obviously, retaining the changes cause extra memory consumption and hence affect the Time Complexity.

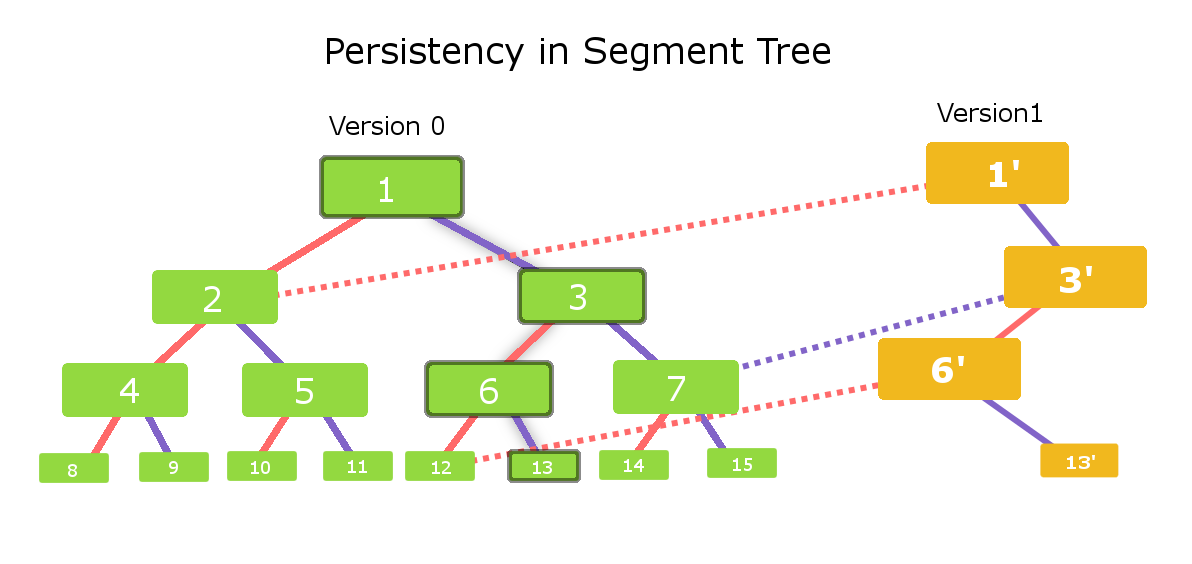
Our aim is to apply persistency in segment tree and also to ensure that it does not take more than **O(log n) time and space** for each change.

Let’s think in terms of versions i.e. for each change in our segment tree we create a new version of it.

We will consider our initial version to be Version-0. Now, as we do any update in the segment tree we will create a new version for it and in similar fashion track the record for all versions.

But creating the whole tree for every version will take O(n log n) extra space and O(n log n) time. So, this idea runs out of time and memory for large number of versions.

Let’s exploit the fact that for each new update(say point update for simplicity) in segment tree, At max logn nodes will be modified. So, our new version will only contain these log n new nodes and rest nodes will be the same as previous version. Therefore, it is quite clear that for each new version we only need to create these log n new nodes whereas the rest of nodes can be shared from the previous version.



Consider the segment tree with green nodes . Lets call this segment tree as **version-0**. The left child for each node is connected with solid red edge where as the right child for each node is connected with solid purple edge. Clearly, this segment tree consists of 15 nodes.

Now consider we need to make change in the leaf node 13 of version-0.   
So, the affected nodes will be – **node 13 , node 6 , node 3 , node 1**.   
Therefore, for the new version **(Version-1)** we need to create only these **4 new nodes**.

Now, lets construct version-1 for this change in segment tree. We need a new node 1 as it is affected by change done in node 13. So , we will first create a new**node 1′**(yellow color) . The left child for node 1′ will be the same for left child for node 1 in version-0. So, we connect the left child of node 1′ with node 2 of version-0(red dashed line in figure). Let’s now examine the right child for node 1′ in version-1. We need to create a new node as it is affected . So we create a new node called node 3′ and make it the right child for node 1′(solid purple edge connection).

In the similar fashion we will now examine for **node 3′**. The left child is affected , So we create a new node called **node 6′** and connect it with solid red edge with node 3′ , where as the right child for node 3′ will be the same as right child of node 3 in version-0. So, we will make the right child of node 3 in version-0 as the right child of node 3′ in version-1(see the purple dash edge.)

Same procedure is done for node 6′ and we see that the left child of node 6′ will be the left child of node 6 in version-0(red dashed connection) and right child is newly created node called **node 13′**(solid purple dashed edge).  
Each **yellow color node** is a newly created node and dashed edges are the inter-connection between the different versions of the segment tree.

**d)Two dimensions segment tree:**

Given a rectangular matrix **M[0…n-1][0…m-1]**, and queries are asked to find the sum / minimum / maximum on some sub-rectangles **M[a…b][e…f]**, as well as queries for modification of individual matrix elements (i.e **M[x] [y] = p** ).

We can also answer sub-matrix queries using Two Dimensional Binary Indexed Tree.

We will focus on solving sub-matrix queries using two dimensional segment tree.Two dimensional segment tree is nothing but segment tree of segment trees.

**Algorithm :**  
We will build a two-dimensional tree of segments by the following principle:   
**1 .**In First step, We will construct an ordinary one-dimensional segment tree, working only with the first coordinate say ‘x’ and ‘y’ as constant. Here, we will not write number in inside the node as in the one-dimensional segment tree, but an entire tree of segments.   
**2.**The second step is to combine the values of segmented trees. Assume that in second step instead of combining the elements we are combining the segment trees obtained from the step first.

**5/Quiz about Segment tree:**

**1 .**What is Segment Tree ?

* 1. Tree
  2. Grapth
  3. Array
  4. Variable

**2 .**What is the height of the segment tree?

* 1. O(logn),
  2. O(n),
  3. O(n^2),
  4. O(n\*logn),

**3 .**How many elements does the leaf of a segment tree contain?

* 1. 1
  2. 2
  3. 4
  4. 8

**4 .**How many elements does the root of a segment tree contain?

* 1. N/2
  2. N
  3. N/4
  4. N/8

**5.**What is the main operation of segment tree:

* 1. Increase number of node
  2. Update tree
  3. Decrease number of node
  4. Union 2 segment tree

**6.**Can simple linear array be used to represent the Segment Tree

* 1. Yes
  2. No
  3. Yes but number of node need to be odd
  4. Yes but all of node need to be positive

**7.** What is the complexity of update segment tree classically?

* 1. O(logn),
  2. O(n),
  3. O(n^2),
  4. O(n\*logn),.

**8.** What is the complexity of query segment tree classically?

* 1. O(logn),
  2. O(n),
  3. O(n^2),
  4. O(n\*logn),

**9.** Technique that can be used to reduce the complexity of a segment tree is:

* 1. Union
  2. Heap
  3. Queue
  4. Lazy propagation

**10.** Why we need to use segment tree?

* 1. For faster
  2. Because like
  3. Look like pro programmer
  4. Because it’s fun