



how long will you take to code a segment tree from scratch

Generic Segment Trees

20 min for complete code

10 min for modifying an existing template

- Priyansh Agarwal

What is a Generic Segment Tree



- A generic segment tree is a segment tree template that can be modified to fit any use case by making the minimum number of changes.
- It separates out the static part and the dynamic part making it easier to make edits where it is relevant. Eg: going down the tree for an update query is static in all segment trees but what happens on the leaf node in that update is dynamic.
- It also allows us to create multiple segment trees in the same code without having to rewrite the static part more than once.

* also helps avoid many typos and logical errors

Generic Segment Tree Template:

https://github.com/Priyansh19077/CP-Templates/blob/master/Range%20Queries/Segment_tree.cpp



Generic Segment Tree Node Struct



how to merge two children to set val of parent

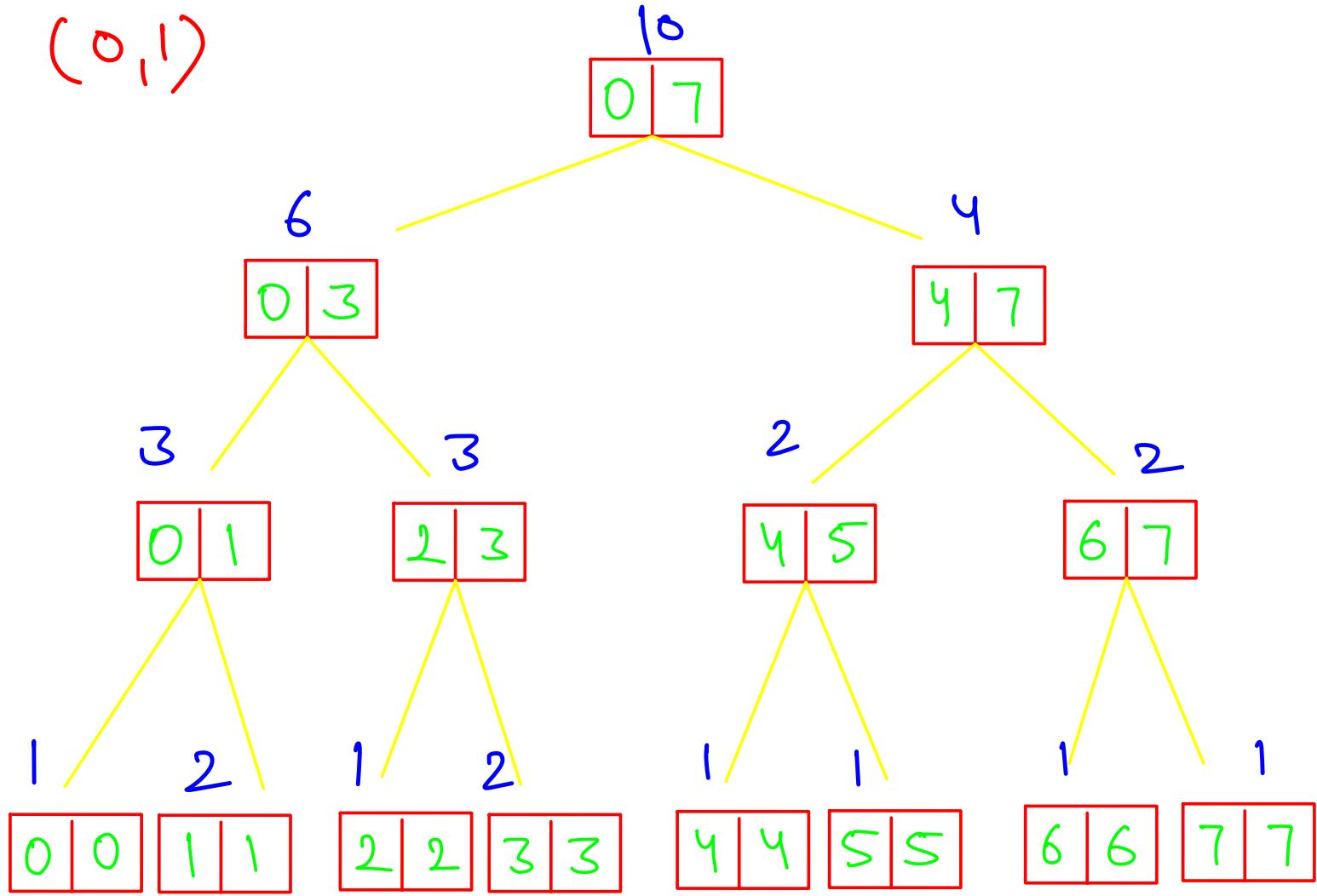
```
struct Node1 {  
    ll val; // may change  
    Node1() { // Identity element  
        val = 0; // may change  
    }  
    Node1(ll p1) { // Actual Node  
        val = p1; // may change  
    }  
    void merge(Node1 &l, Node1 &r) { // Merge two child nodes  
        val = l.val + r.val; // may change  
    }  
};
```

what we store
in every segment
tree node must be
modifiable

what to return
when there is no overlap
(default value)

how to populate all the
information needed from the array index

query → (0,1)



Generic Segment Tree Constructor



```
template<typename Node, typename Update>
struct SegTree {
    vector<Node> tree;
    vector<ll> arr; // type may change
    int n;
    int s;
    SegTree(int a_len, vector<ll> a) { // change if type updated
        arr = a;
        n = a_len;
        s = 1;
        while(s < 2 * n){
            s = s << 1;
        }
        tree.resize(s); fill(all(tree), Node());
        build(0, n - 1, 1);
    }
}
```

$$\begin{aligned}n &= 15 \rightarrow 4n = 60 \\&\rightarrow \underline{\underline{31}} \quad \underline{\underline{32}}\end{aligned}$$

$$\text{nodes}(n) \leq \text{nodes}(2^k)$$

such that

$$2^k \text{ is the first power of } 2 > n$$

$$2 \cdot [\text{first power of } 2 \geq n] - 1$$

$$\text{first power of } 2 > 2n$$

$$\rightarrow \text{nodes}(n) < 4n$$

$$n = 9 \rightarrow 36$$

$$2^k = 16$$

$$2(2^k) - 1 = 31$$

Generic Segment Tree Build Function



```
        build(0, n - 1, 1);
    }

void build(int start, int end, int index) // Never change this
{
    if (start == end)      {           leaf node
        tree[index] = Node(arr[start]);
        return;
    }
    int mid = (start + end) / 2;
    build(start, mid, 2 * index);
    build(mid + 1, end, 2 * index + 1);
    tree[index].merge(tree[2 * index], tree[2 * index + 1]);
}
```

↑ parameterized constructor

↑ merge function of node

update \rightarrow $a[i] \rightarrow (\underline{a[i].x} + y)$

Generic Segment Tree Update Struct



```
struct Update1 {  
    ll val; // may change  
    Update1(ll p1) { // Actual Update  
        val = p1; // may change  
    }  
    void apply(Node1 &a) { // apply update to given node  
        a.val = val; // may change  
    }  
};
```

pars as many variables
as we want

the way in which we
update the leaf node

leaf node

Generic Segment Tree Update Function



```
void update(int start, int end, int index, int query_index, Update &u)
{
    if (start == end) {
        u.apply(tree[index]);
        return;
    }
    int mid = (start + end) / 2;
    if (mid >= query_index)
        update(start, mid, 2 * index, query_index, u);
    else
        update(mid + 1, end, 2 * index + 1, query_index, u);
    tree[index].merge(tree[2 * index], tree[2 * index + 1]);
}
```

→ update's function

→ node's function

never change

```
void make_update(int index, ll val) { // pass in as many parameters as required
    Update new_update = Update(val); // may change
    update(0, n - 1, 1, index, new_update);
}
```

User will call it



Generic Segment Tree Query Function

```
Node query(int start, int end, int index, int left, int right) {  
    if (start > right || end < left)           no overlap  
        return Node();  
    if (start >= left && end <= right)          complete overlap  
        return tree[index];  
    int mid = (start + end) / 2;  
    Node l, r, ans;  
    l = query(start, mid, 2 * index, left, right);  
    r = query(mid + 1, end, 2 * index + 1, left, right);  
    ans.merge(l, r);  
    return ans;  
}
```

```
Node make_query(int left, int right) { ←  
    return query(0, n - 1, 1, left, right);  
}
```

Problems



- ✓ Range Min
- ✓ Range Xor
- ✓ Range GCD



More Interesting Problems

- Range Second Min
- Range Min Index
- First element > X in Range

[1 | 2 | 1 | 2 | 2 | 2 | 3]

[1 | 2 | 4 | 5 | 7 | 9]

2nd min

what to store in leaf $\rightarrow \{\underline{\text{val}}, \text{inf}\}$

merge left and right

left
↳ 1st min
2nd min

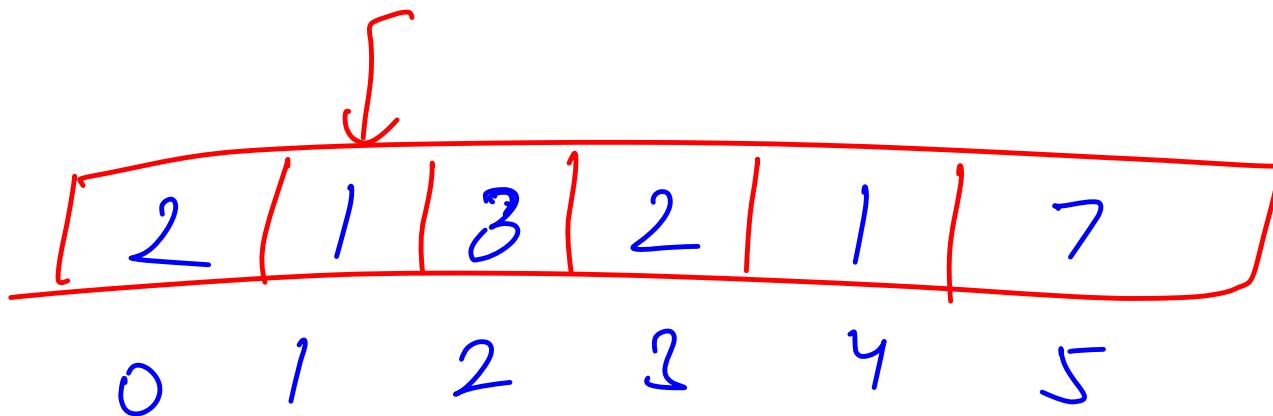
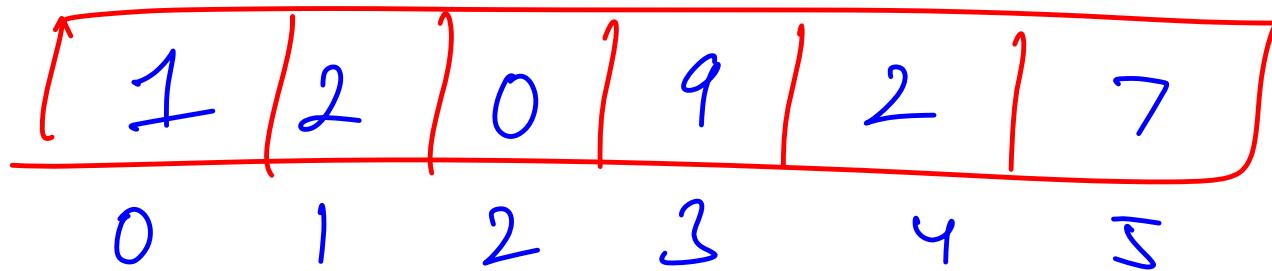
1st min
right
2nd min

$[l_1 \ l_2 \ r_1 \ r_2] \rightarrow \text{soot}$

1st min = arr[0] 2nd min = arr[1]

Default value $\rightarrow \{\text{cat}, \text{inf}\}$

range min index (return index of 1st min if there are multiple instances of the min in the range)



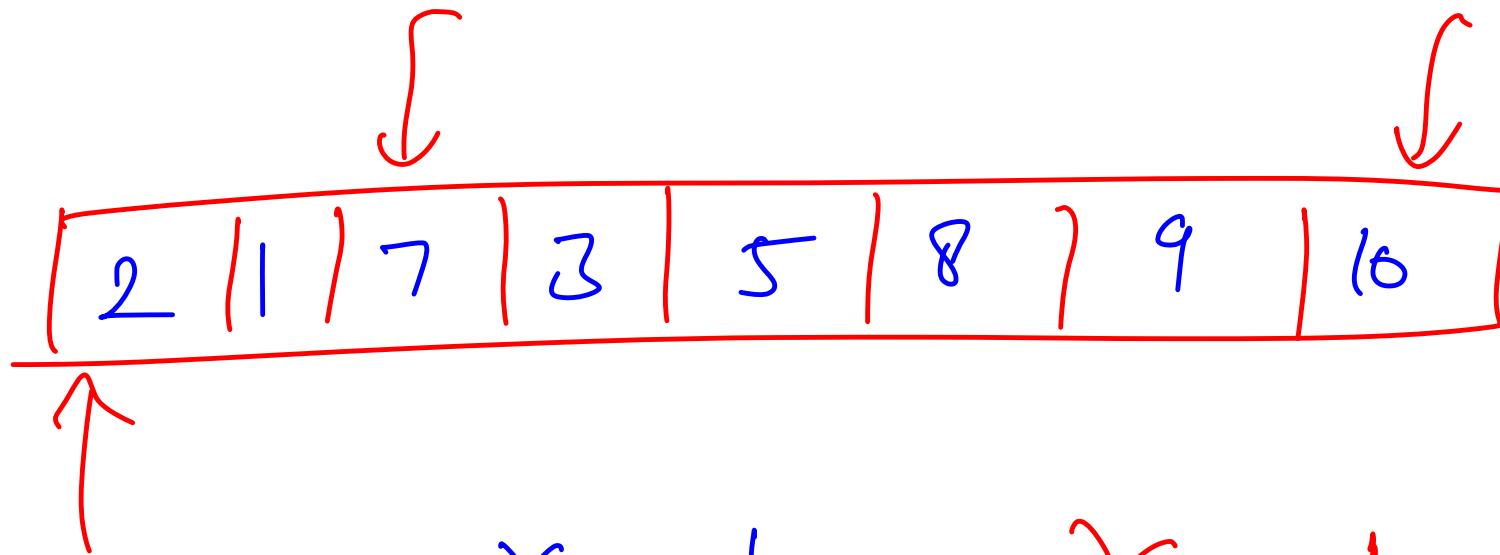
store → val, index

merge → L.val, L.index R.val, R.index
if (L.val ≤ R.val) → L

else \rightarrow R anything

Default value $\rightarrow \{ \inf, \underline{\inf} \}$
won't matter

1st element $> X$ in the range



$$X = 4$$

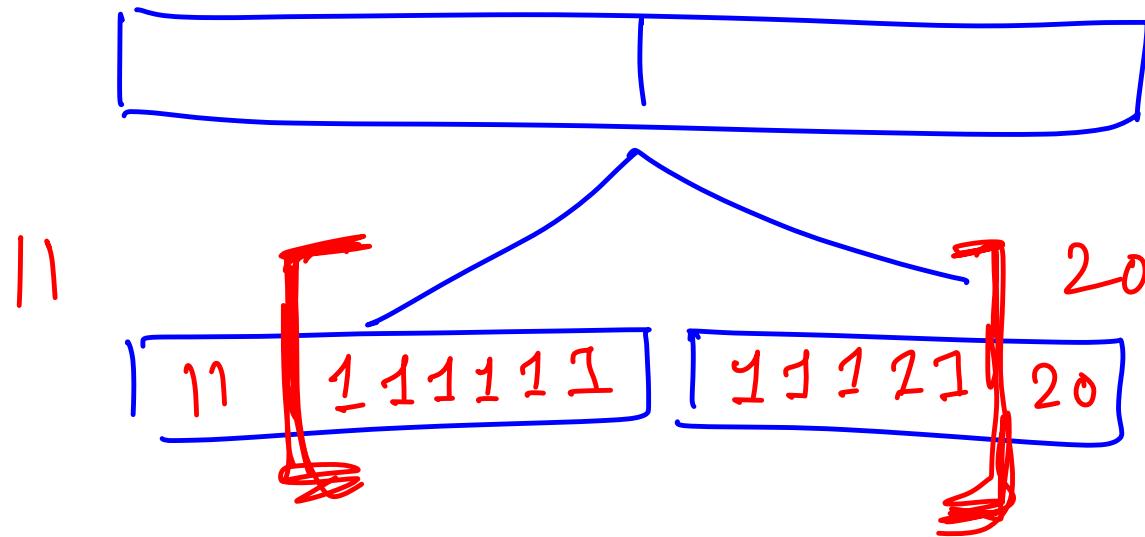
=

$$X = 9$$

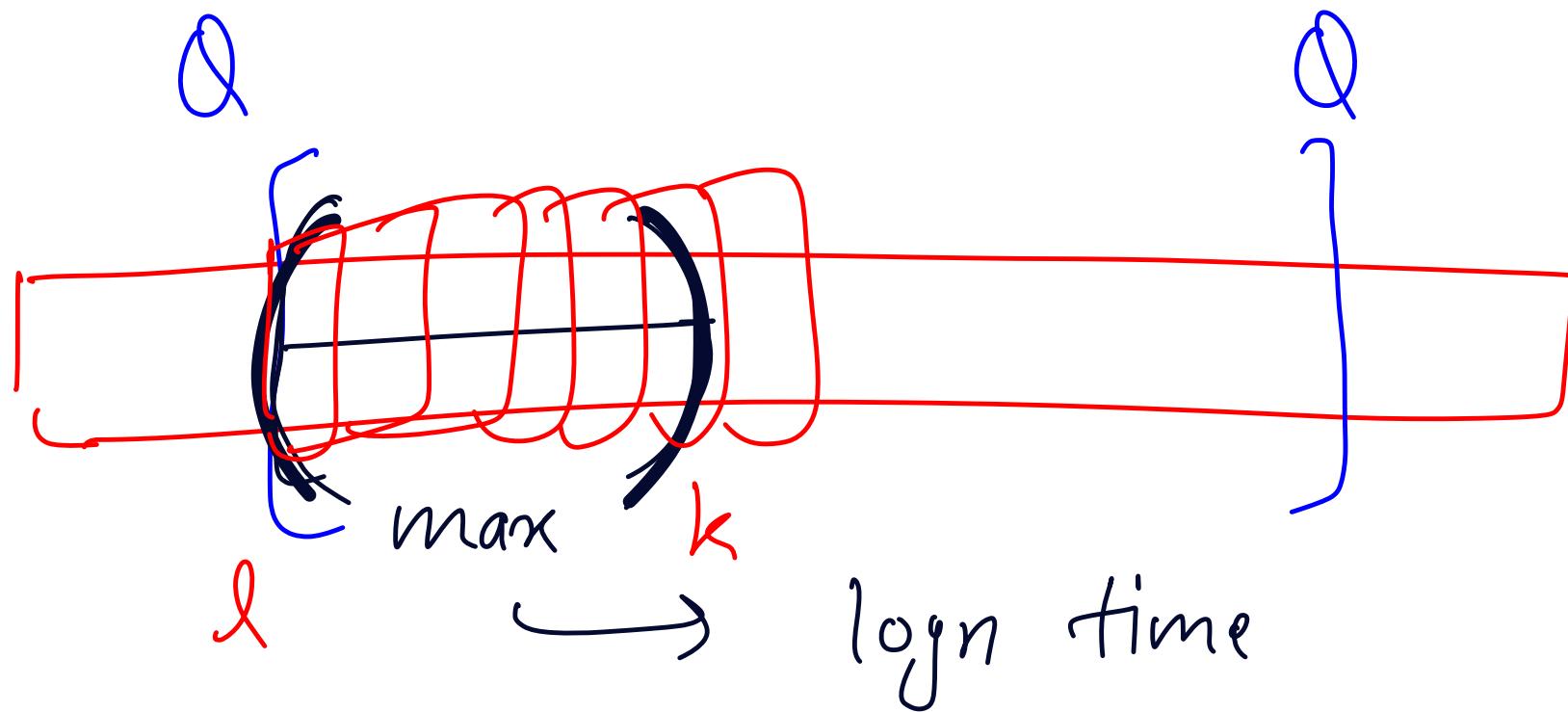
$$X = 1$$

store maximum at every node

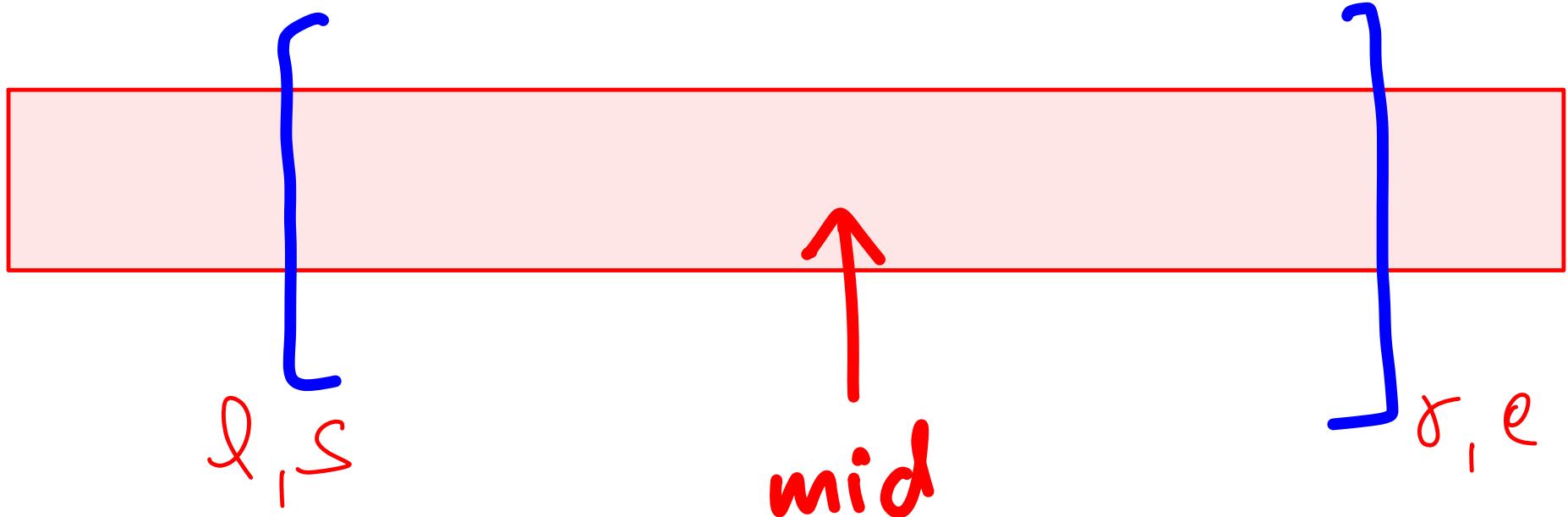
$$x = 18$$



Store the maximum at every node $x = 20$



$$\max(l \text{ to } k) < \max(l \text{ to } k+1)$$



if (~~sq.make-query(l, mid) > x~~)

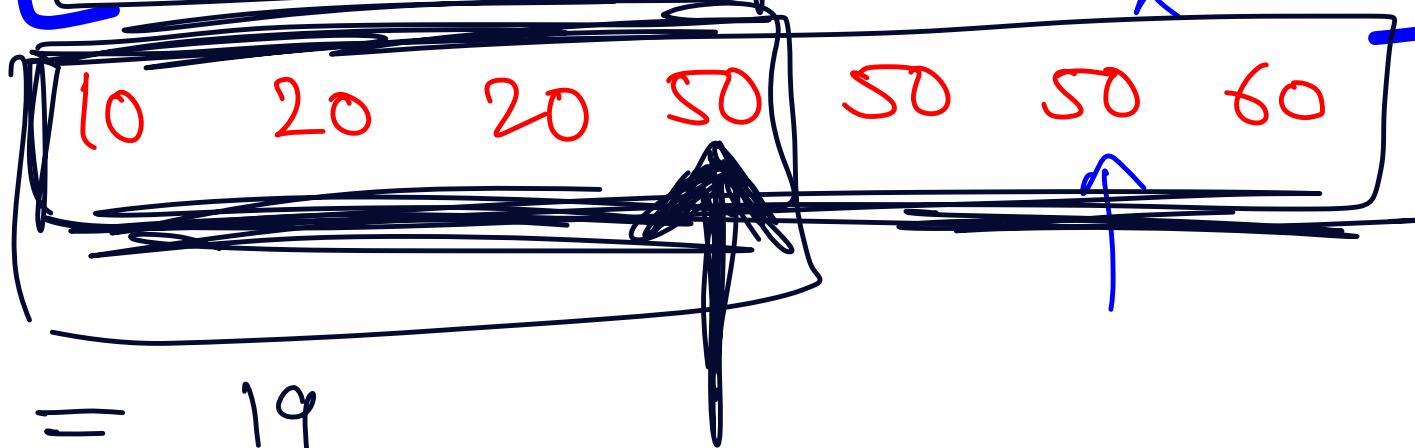
ans = mid ,
 $e = mid - 1$

else

$s = mid + 1$

(00 2	[10 20 5 50	40 20]	60	100
-------	-------------	--------	----	-----

$p_{xt,max}$



$$x = 19$$

$$=$$

$$x = 55$$