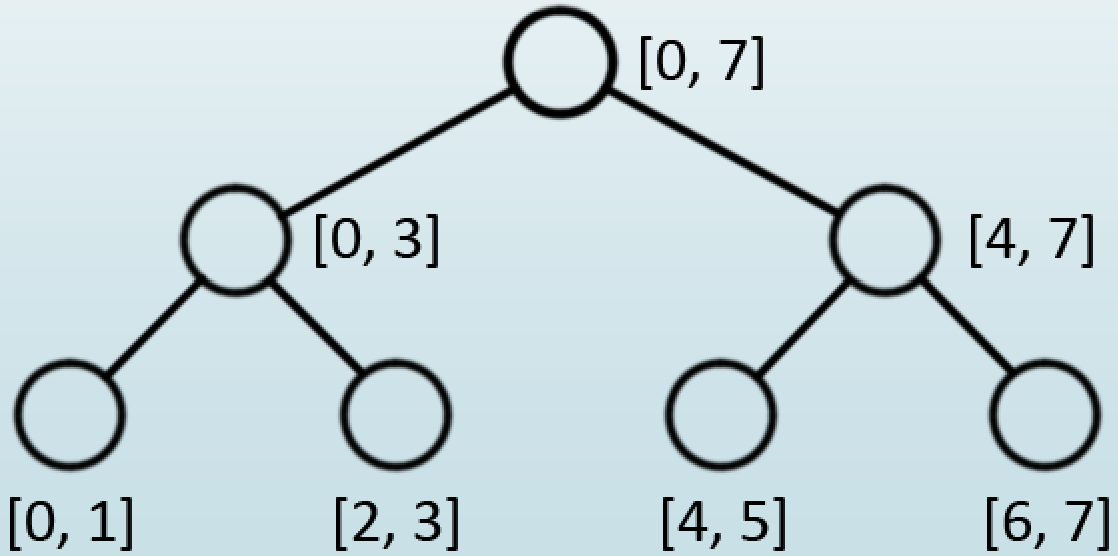
Segment Tree

Pittsford Sutherland Programming Club

By Yiyou Chen, Yizuo Chen

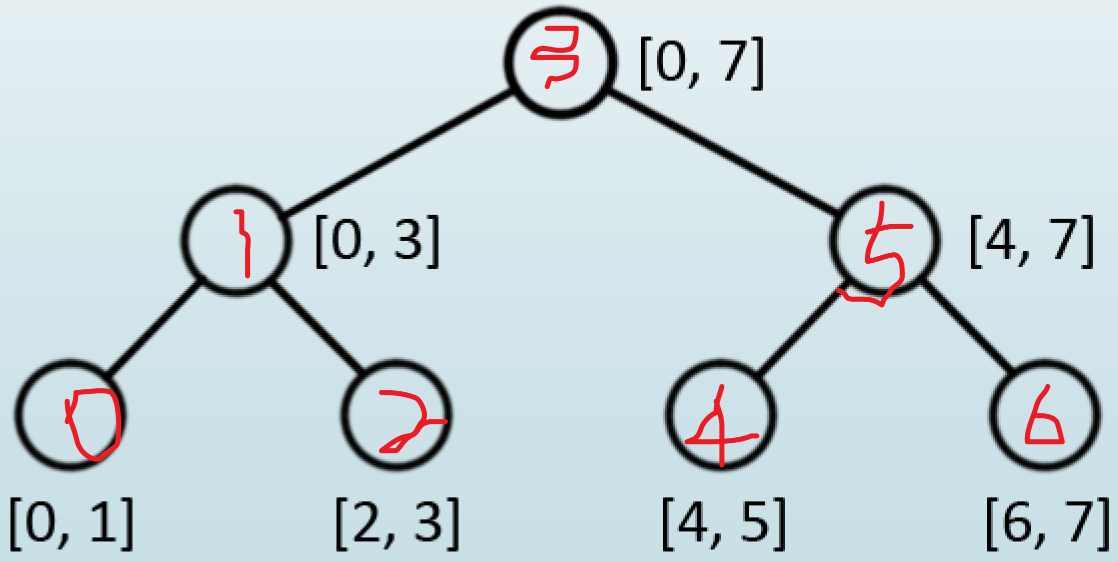
Segment tree

- Segment tree is a tree data structure for storing intervals, or segments.
- Each node of the tree is has an interval.



Segment tree

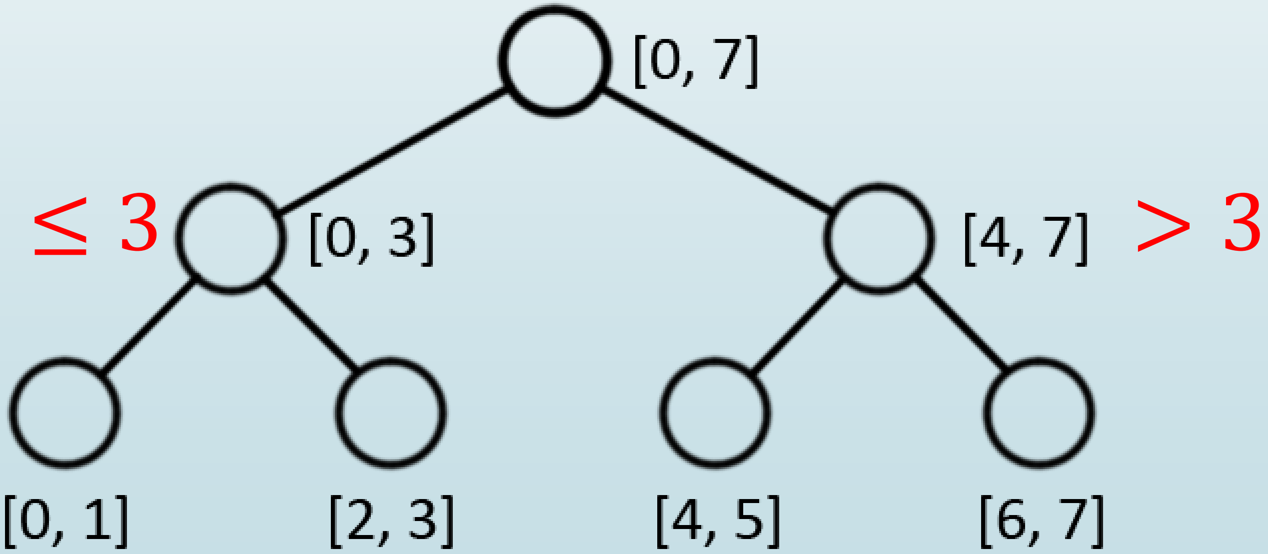
- Intervals [left, right] in **black**
- mid = (left + black) / 2 are in **red**



Segment tree

A node's left subtrees are the intervals less or equal to the node's mid value

A node's right subtrees are the intervals greater than the node's mid value



Elements in each node

- Struct NODE{

```
int left;        // [l, r]  -> l
int right;       // [l, r]  -> r
int mid;         //  (left + right) / 2;
int val; // usually maximum or sum
```

- }tree[size];

Segment Tree

- Rank of left son = rank of parent * 2;
- Rank of right son = rank of parent * 2 + 1;

Example: To build a segment tree to store the sum of the intervals

Build a segment tree – up down

```
build_tree(int left, int right, int root, int a[1024], NODE *tree){
    tree[root].l = left;                        // create a new node
    tree[root].r = right;                      // create a new node
    tree[root].mid = (tree[root].l + tree[root].r) / 2;   // calculate the mid value
    if(left == right){                                     // if it’s a leaf
        tree[root].val = a[left];
        return;
    }
    build_tree(left, tree[root].mid, root * 2, a, tree);       //update its left sub-tree
    build_tree(tree[root].mid + 1, right, root * 2 + 1, a, tree); // update right sub-tree
    tree[root].val = tree[root * 2].val + tree[root * 2 + 1].val;  // update parent value
```

update a specific element

- Step1: find the position of the element
- Step2: update the value of the element
- Step3: update the value of its parent interval.

update a specific element

```
update_tree(int root, int id, int add, NODE *tree){
    tree[root].val += add;          // update the element’s value
    if(tree[root].l == tree[root].r)
        return;                          // it is a leave
    if(id <= tree[root].mid)
        update_tree(root * 2, id, add, tree);      // update LT
    else update_tree(root * 2 + 1, id, add, tree);  // Update RT
}
```

Update an interval

- Step1: Find the interval.
- Step2: Combine the intervals.

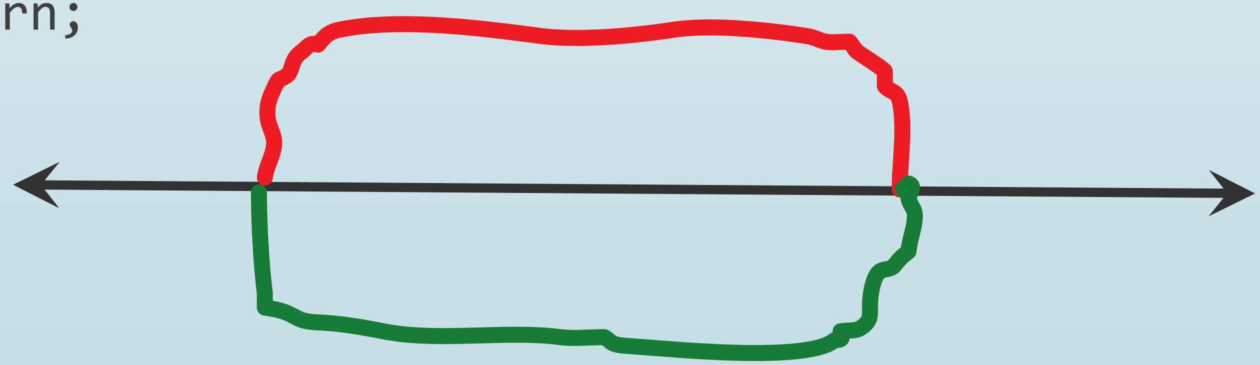
Step1: Find the interval – interval analysis

- The interval on the tree is in Red.
- The interval we need to find in Green.
- There are three cases.

Case1: We get lucky!

- The interval we want to find is exactly the same as the interval on the tree. Directly update the interval!

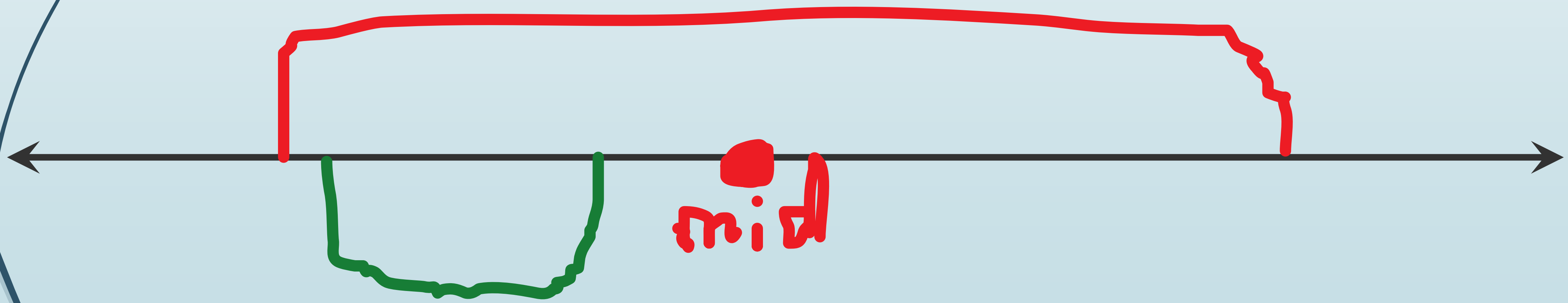
```
if(tree[root].l == left && tree[root].r == right){
        tree[root].val += add;
        update(root * 2, left, mid, add);
        update(root * 2 + 1, mid + 1, right, add);
        return;
}
```



Case 2: the interval we want is on the left of mid point!

- We need to search the interval we want on its left sub-tree.

```
if(right <= tree[root].mid){
        update_interval_tree(root * 2, left, right, add, tree);
}
```



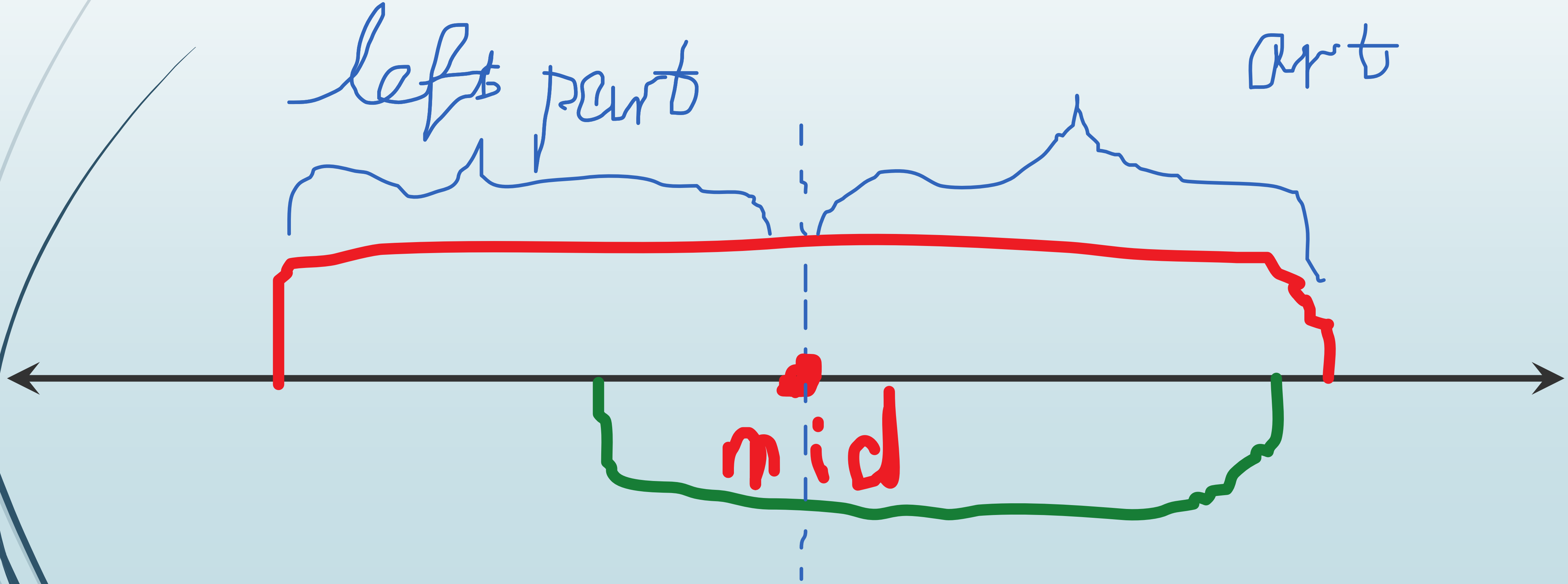
Case 3: the interval we want is on the right of mid point

- We need to search the interval we want on its right sub-tree.

```
else if (left > tree[root].mid){
        update_interval_tree(root * 2 + 1, left, right, add, tree);
}
```

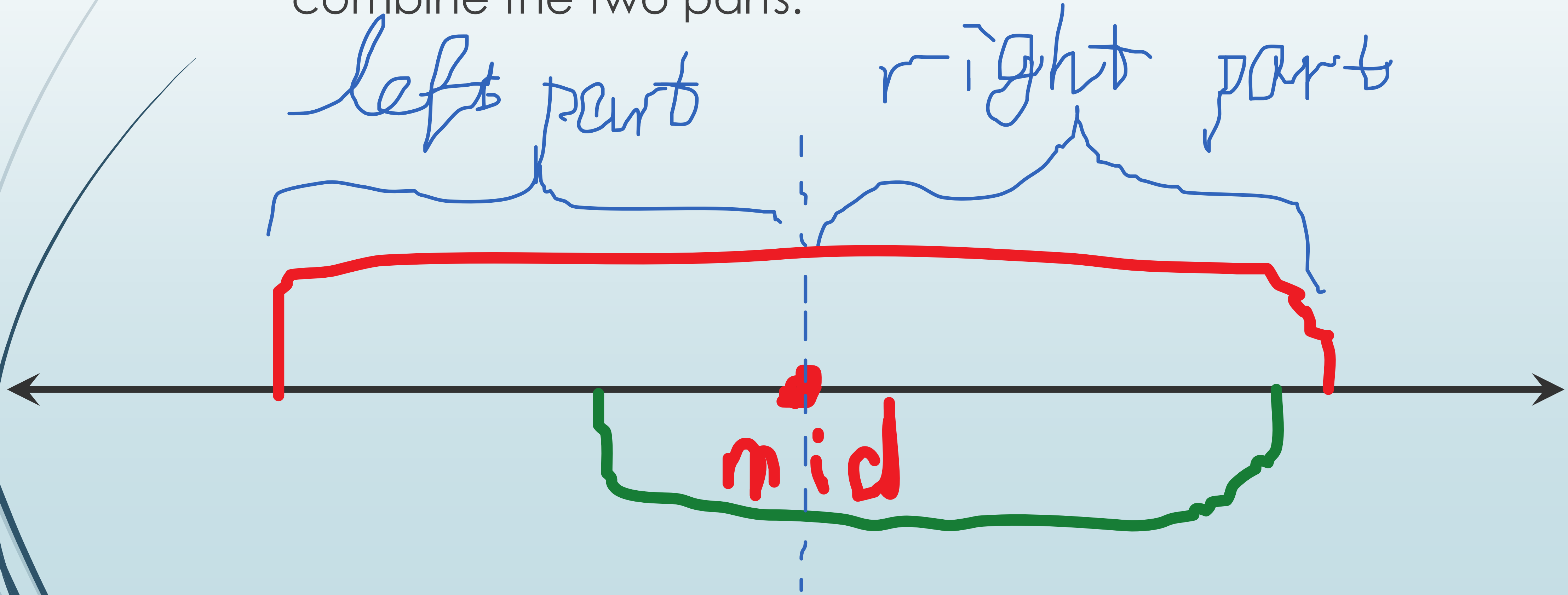
Case 4: the interval we want is cut into two parts by the mid point

- The mid point of the interval on tree is on the interval we want to find.



Case 4: the interval we want is cut into two parts by the mid point

- Search the left part and right part independently, then combine the two parts.



How to Combine?

In order to get the sum of the interval, we can simply combine the two parts by adding the two values together.

```
else{
        update_interval_tree(root * 2, left, tree[root].mid, add, tree);
        update_interval_tree(root * 2 + 1, tree[root].mid + 1, right, add,
tree);
}

tree[root].val = tree[root * 2].val + tree[root * 2 + 1].val;
```

How to Combine?

- Combine is the hardest part of segment tree.

Some common examples:

```
Max number: Tree[x].val = max(tree[x * 2].val, tree[x * 2 + 1].val);
Min number:  tree[x].val = min(tree[x * 2].val, tree[x * 2 + 1].val);
```

Segment tree query (find the sum on a certain interval)

- Same as the update process:
- 1. find the interval (see slide 13 - 17)
- 2. return the value

Time complexity

- Consider an array a[1 -- n] with n elements. How many nodes are there on a segment tree if we construct a segment tree based on the array?
- Ans: 2n – 1. All the elements in array a will become leaves on the segment tree. Assume the height of the tree is h, the 0 level has

  2^0 nodes, the 1st level contains 2^1 nodes, and level h contains 2^h nodes. The total nodes from level 0 to level h contains $2^{h+1} - 1$ nodes.

Since $2^h = n,$ $\quad total\ number\ of\ nodes = 2n - 1$

Time complexity for building a segment tree O(n)

- Since we need to add every elements in n to the tree, it takes O(n) time to construct a segment tree.

Time complexity for updating a single node on segment tree O(logn)

- In order to find a certain node on the tree, we only need to search through the height of the tree which costs O(logn) time.

Time complexity for querying an interval on segment tree O(logn)

- Divide a large interval into two parts all the time during the searching process, so it only goes through the height of the tree, which takes O(logn) time to query an interval.

Time complexity for updating an interval on segment tree O(n) (Naïve)

Whenever we find the interval we want to update, we update the value of all its sub-intervals, which takes O(n) time.

In order to make this operation take O(logn) time, we need to use

Lazy propagation!

Lazy propagation

- After finding the interval we want to update, can we just give it a mark says "add x to this interval"? So we don't necessary change all its sub-intervals' values.
- Then when we query the sum of the interval, we can then add the lazy value to the sum as we go through the intervals.
- We're trying to become as "lazy" as possible that we just give a mark on the interval we want to update instead of updating all its children.

Update an interval with lazy propagation

```
if(tree[root].l == left && tree[root].r == right){   // find the interval
        tree[root].lazy += add;              // give a mark
        return;
    }
    tree[root].val += add * (right - left + 1);
   // since all the sons will be added same value, the total value we need to add
on the root is add * (number of its children and itself)
```

Query with lazy propagation

```
if(left == tree[root].l && right == tree[root].r){
        return tree[root].lazy * (right - left + 1) + tree[root].val; //update
    }
    if(tree[root].lazy){      // update its children
        tree[root].val += tree[root].lazy * (tree[root].r - tree[root].l + 1);
        tree[root * 2].lazy += tree[root].lazy;
        tree[root * 2 + 1].lazy += tree[root].lazy;
        tree[root].lazy = 0;
    }
```

Time Complexity for update with Lazy Propagation O(log n)

- For the interval update, it takes only O(log n) time to find the interval, then update the interval.
- For the query process, it's takes the same time as the one without lazy propagation.