

## Data Structures II

### COMP4128 Programming Challenges

School of Computer Science and Engineering  
UNSW Australia

## Table of Contents

2

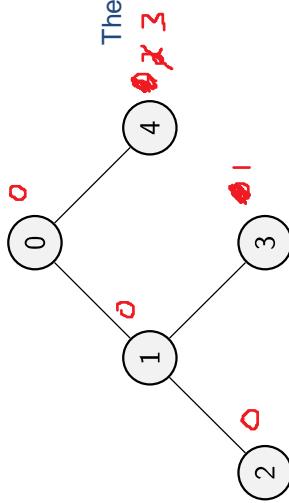
- 1 Range Trees over Trees
  - Range Trees over Trees
  - Range Updates, Point Queries
  - Range Updates, Range Queries
  - Range Updates, Range Queries
  - Range Trees of Data Structures
  - Solving Problems in Subranges
  - Searching a Range Tree

- For this section, assume all trees are rooted with a specified root.
- We've seen how to do certain path queries. Another natural and useful question is how to do subtree queries/updates. a subtree is really just a range

- **Problem Statement** Given a tree rooted at node 0, each node has a value, all values are initially 0. Support the following 2 operations.
  - **Update:** Of the form  $U \ a_i \ w_i$ . Change the value of node  $a_i$  to  $w_i$ .
  - **Query:** Of the form  $Q \ a_i$ . What is the sum of values in the subtree rooted at vertex  $a_i$ ?
- **Input** First line,  $V, Q$ , number of vertices and operations.  
 $1 \leq V, Q \leq 100,000$ . The next line specifies the tree.  
 $V - 1$  integers,  $p_i$ , the parent of vertex  $i$  (1-indexed).  
 The following  $Q$  lines describe the updates and queries.  
 $1 \leq V, Q \leq 100,000$ .
  - **Output** For each **Query**, an integer, the sum of values in the subtree rooted at  $a_i$ .

Range Tree on Trees

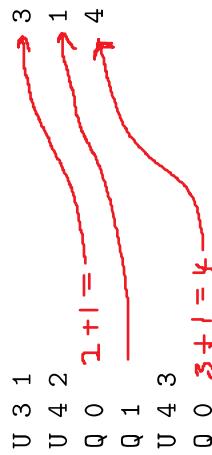
5



## Sample Queries:

## Sample Output:

The numbers inside the nodes are not the values of the tree, they are the index of the nodes as seen when doing a DFS.



- To support general subtree queries, we will extend our range trees to work on trees.
  - The key here is to find an ordering of the vertices such that every subtree corresponds to a range of indices.
  - Actually, any sensible DFS ordering already does this.
- In data structures 1, we saw range trees that work on arrays. We can make range trees work on trees if we simply write the tree as an array. We cannot use the usual root at 0, children at  $2i+1$  and  $2i+2$  though, because elements of a subtree must be contiguous in the array (because in range trees on arrays the ranges must be contiguous in the array).
- We can get contiguous subtrees in array form by simply doing a DFS.
- DFS processes all nodes in a subtree before returning from the subtree. So as long as we're assigning ids consecutively, a whole subtree should get consecutive indices.

- **Implementation:** In your DFS that creates a representation of the tree, also either preorder or postorder the vertices. Each node should store the range of indices that exists in its subtree.
- Now build your range tree over these indices. Past this point, you can forget about your tree and just work on your array of indices.
- To update node  $u$ , look up what its index is. Then just update your range tree at  $indexInRangeTree[u]$ .
- To query a subtree rooted at  $v$ , look up the range of indices in its subtree. Then just query your range tree for the range  $[startRange[v], endRange[v]]$ .
- **Moral:** Queries on subtrees are essentially the same as just normal range queries.

```

Data Structures II

Range Trees over Trees
Range Updates
Point Queries
Range Updates
Range Queries
Range Trees of Data Structures
Solving Problems in Subranges
Searching a Range Tree

#include <bits/stdc++.h>
using namespace std;

const int MAXN = 100000;
// Suppose you already have your tree set up.
vector<int> children[MAXN+5];
// A node is responsible for the range [startRange[v], endRange[v])
int indexInRangeTree[MAXN+5], startRange[MAXN+5], endRange[MAXN+5];
int rotId;
// A range tree that supports point update, range sum queries.
long long rangeTree[4*MAXN+5];
void update(int plc, int v);
long long query(int qL, int qR); // Query for [qL, qR)

void computeTreeRanges(int c) {
    indexInRangeTree[c] = startRange[c] = totId++;
    for (int nxt : children[c]) {
        computeTreeRanges(nxt);
    }
    endRange[c] = totId;
}

void updateNode(int id, int v) {
    update(indexInRangeTree[id], v);
}

int querySubtree(int id) {
    return query(startRange[id], endRange[id]);
}

int queryRange(int startRange, int endRange) {
    return query(querySubtree(startRange), querySubtree(endRange));
}

```

## Table of Contents

9

### Data Structures II

- 1 Range Trees over Trees
- 2 **Range Updates, Point Queries**
- 3 Range Updates, Range Queries
- 4 Range Trees of Data Structures
- 5 Solving Problems in Subranges
- 6 Searching a Range Tree

### Range Trees over Trees

### Range Updates, Point Queries

### Range Updates, Range Queries

### Range Trees of Data Structures

### Solving Problems in Subranges

### Searching a Range Tree

## Range Updates, Point Queries

10

- point updates involved traversing from the root down to the leaf node, updating the leaf node and then propagating this change back up to the root.
  - range queries involved only going as deep as required to fully cover a range, at which point the result for that entire range is stored at.
  - now we want to change updates such that we aren't just updating one leaf node, but many. Of course we cannot actually just update all the leaf nodes specified directly, because this is implicitly  $O(n)$ . Instead, we leverage the same technique as for range queries.
- Recall range trees. They were a data structure that for many types of operations supported range queries and point updates.
  - We will now extend this to also support range updates.
  - For simplicity, let us tackle the problem of range updates, point queries first.

- Given an array  $a[N]$ , initially all zeros, support  $Q$  operations, each being one of the following forms:
  - Update:**  $U \ i \ r \ v$ . Perform  $a[i], \dots, r) += v$ .
  - Query:**  $Q \ x$ . Output  $a[x]$ .
- $N, Q$  is up to 100,000.

range update, must also be done in  $\log(n)$   
point query done in  $\log(n)$

- Again, we can't just individually update all elements of the array, that would cost  $O(N)$  per operation.
- So we are going to do the same as we did for range queries.
- Suppose our update tells us to perform  $a[1, r) += x$ .
- This is the same as performing  $a[1, m) += x$  and  $a[m, r) += x$ .
- So we can partition our initial update into smaller ranges however we wish.

- We will decompose  $[l, r)$  into ranges that correspond directly to nodes in our range tree in the same way that we do for range queries.
  - For each node we will store a *lazy counter* that keeps the sum of all updates to that node's range of responsibility.
  - To query an index, we need to know all updates to ranges that contain said index.
  - For a range tree there are  $O(\log n)$  of these ranges, and they are exactly the ranges that appear on the path from the root to the leaf corresponding to that index.
- i.e. if you store the lazy update on the parent of the subtree, then if you query for the value of any node in that subtree, then you will have to pass through that parent, at which point you can accumulate that lazy update down into that tree.

- Let's update the range [2, 8) with  $v = 3$ .

Range Updates

Joint Quantiles

Range Queries

Strange Cases  
of Data  
Structures

Solving Problems in

0 [0, 8) [2, 8) 2..8 is not a match, push it down

```

graph LR
    P1((Player 1)) --> C1_1[0 [4, 8]]
    P1 --> C1_2[0 [4, 6)]
    P1 --> C1_3[0 [0, 4)]
    C1_1 --> C2_1_1[0 [0, 2)]
    C1_1 --> C2_1_2[0 [2, 4)]
    C1_2 --> C2_2_1[0 [0, 0)]
    C1_2 --> C2_2_2[0 [0, 0)]
    C1_3 --> C2_3_1[0 [0, 0)]
    C1_3 --> C2_3_2[0 [0, 0)]
    C2_1_1 --> C3_1_1_1[0 [0, 0)]
    C2_1_1 --> C3_1_1_2[0 [0, 0)]
    C2_1_2 --> C3_1_2_1[0 [0, 0)]
    C2_1_2 --> C3_1_2_2[0 [0, 0)]
    C2_2_1 --> C3_2_1_1[0 [0, 0)]
    C2_2_1 --> C3_2_1_2[0 [0, 0)]
    C2_2_2 --> C3_2_2_1[0 [0, 0)]
    C2_2_2 --> C3_2_2_2[0 [0, 0)]
    C2_3_1 --> C3_3_1_1[0 [0, 0)]
    C2_3_1 --> C3_3_1_2[0 [0, 0)]
    C2_3_2 --> C3_3_2_1[0 [0, 0)]
    C2_3_2 --> C3_3_2_2[0 [0, 0)]

```

ooal values:

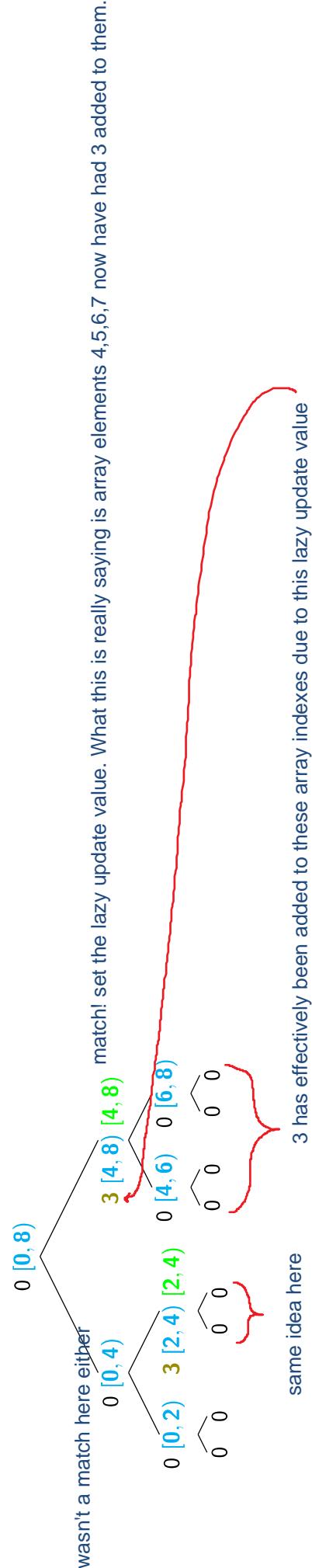
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
3	3	3	3	3	3	3	3

the array is initially all 0's, but now we want to update the range 2..8 to be 3

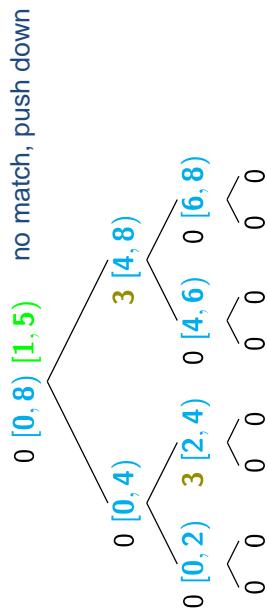
## Range Updates, Point Queries

15

- As with range queries, we will push the update range down into the applicable nodes.

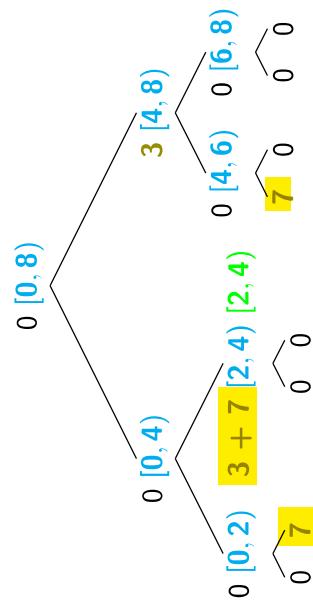


- Now let's assume we've been given a second update, for the range  $[1, 5)$  with  $v = 7$ .

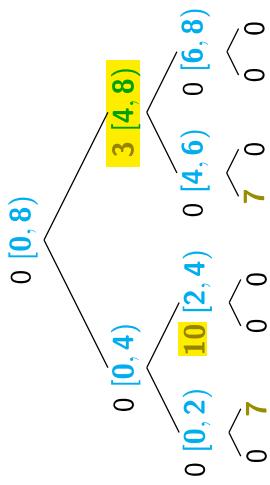


index 1 itself must be updated, because there is no higher range tree node that specifies just node one by itself (one up includes node 0 as well, and this isn't wanted to be updated).  
 indexes 2 and 3 can be updated by range tree node responsible for  $[2, 4)$  as this specified both of those.  
 index 4, like 1, must also be updated solely because we do not want to include array element 5 in the update.

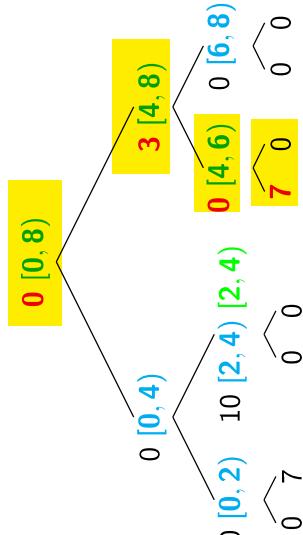
- The range decomposition is  $[1, 2), [2, 4), [4, 5)$ .



- Note that we have not changed the value for the node corresponding to the range [4, 8).



- Let's try to query what  $a[4]$  is. The nodes responsible for a range containing  $i = 4$  are the ones from the leaf for  $i$  to the root.



- The sum of these is  $0 + 3 + 0 + 7 = 10$ , hence  $a[4] = 10$ .

## Range Updates, Point Queries

20

Data

Structures II

```
#define MAX_N 1000000
long long lazyadd[1266666]; // you don't have to store the actual resultant values anywhere, because you walk the tree on a query they are always generated correctly

// The root node is responsible for [0, MAX_N). Update range [uL, uR)
// Compare to range query code.
void update(int uL, int uR, int v, int i = 1, int cLeft = 0, int cRight = MAX_N) {
    if (cLeft == cRight && uR == cRight) {
        lazyadd[i] += v; // if we match the range, add the lazy update value
        int mid = (cLeft + cRight) / 2;
        if (uL < mid) update(uL, min(uR, mid), v, i * 2, cLeft, mid);
        if (uR > mid) update(max(uL, mid), uR, v, i * 2 + 1, mid, cRight);
        if (i == 1) return; // otherwise check if we need to go left, AND whether we need to go right.
    }
}

long long query(int p, int i = 1, int cLeft = 0, int cRight = MAX_N) {
    if (cRight - cLeft == 1) { // if we make it to a leaf node, return the value from it
        return lazyadd[i];
    }
    int mid = (cLeft + cRight) / 2;
    long long ans = lazyadd[i];
    if (p < mid) ans += query(p, i * 2, cLeft, mid);
    else ans += query(p, i * 2 + 1, mid, cRight);
    return ans;
}
```

Range Trees

over Trees

Range Updates,

Point Queries

Range Updates,

Range Queries

Range Trees

of Data

Structures

Solving

Problems in

Subranges

Searching a

Range Tree

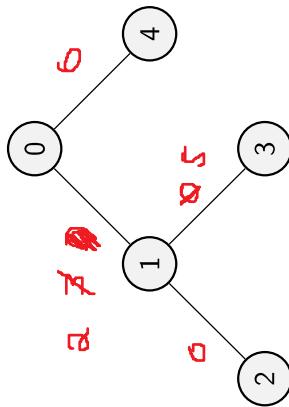
there are default args for cleft and cright because you don't actually pass these in. These are the upper and lower bounds on the range tree ranges (i.e. they start as the bounds that the root node of the range tree has). They then get updated in subsequent recursive calls, depending on which branch of the tree you go down.

- Complexity: Still  $O(\log n)$  per operation, for the same reasons as before.
- Works for most operations that can be broken down into smaller ranges.
- You also need to be able to accumulate the operation so that you can store all the information in the lazy counter.  
So for example, the operation  $a[i] = a[i] \text{ mod } v_q$  is an issue.
- However, covers most operations you would naturally think of. e.g: multiply, divide, xor, and, etc...
- Can also sometimes do multiple different kinds of updates but this is more finicky and depends on the specific updates and probably requires lazy propagation.

- **Problem Statement** Given a weighted tree, all edges initially 0. Support  $Q$  operations, each one taking one of the following forms:
  - **Update**  $U \ a \ b \ w$ . Add  $w$  to the weight of the edge between  $a$  and  $b$ .
  - **Query**  $Q \ a \ b$ : Output the shortest distance between  $a$  and  $b$ .
- **Input** A tree described as  $|V| - 1$  edges. Followed by  $Q$  operations.  $1 \leq |V|, Q \leq 100,000$ .
- **Output** For each query, an integer, the shortest distance from  $a_i$  to  $b_j$ .

## Dynamic Distance on a Tree

23



**Sample Queries:**

U	0	1	3
Q	4	3	
U	0	1	-1
U	3	1	5
Q	3	4	

**Sample Output:**

3  
7

- Recall our solution for the static problem.
- Let  $l := \text{lca}(a, b)$ . We split the path from  $a$  to  $b$  into a path from  $a$  to  $l$  followed by a path from  $b$  to  $l$ .
- Let  $\text{weight\_sum}(a)$  be the sum of weights from the root to  $a$ . The answer is then just  $\text{weight\_sum}(a) + \text{weight\_sum}(b) - 2 * \text{weight\_sum}(l)$ .
- Our updates don't change the tree structure but **change  $\text{weight\_sum}$** . So this is what we need to update.

- When we update an edge, what weight sums do we update?
- Every node whose path to the root goes through said edge. In other words, every node in the edge's subtree.
- So we should maintain weight\_sum using a range tree and update it using subtree updates.

# Dynamic Distance on a Tree

26

Data

Structures II  
#include <bits/stdc++.h>  
using namespace std;

```
Range Trees  
over Trees  
const int MAXN = 100000;  
// Suppose you already have your tree set up.  
int depth[MAXN+5]; // Depth in tree (ignores weight).  
int lca(int a, int b);  
// A node is responsible for the range [startRange[v], endRange[v])  
// in DexRangeTree[MAXN+5], startRange[MAXN+5], endRange[MAXN+5];  
// A range tree supporting range updates of add, point queries of value.  
Range Updates  
Updates  
void update(int uL, int uR, long long v); // value[uL, uR) += v  
Range Queries  
long query(int q);  
void updateEdge(int a, int b, long long v) {  
    // To update the edge's subtree, we need to know which of the 2 nodes  
    // are lower.  
    if (depth[a] > depth[b]) swap(a, b);  
    update(startRange[b], endRange[b], v);  
}  
long long getTreeDistance(int a, int b) {  
    int l = lca(a, b);  
    return query(indexInRangeTree[a]) + query(indexInRangeTree[b])  
        - 2 * query(indexInRangeTree[l]);  
}
```

Searching a  
Range Tree

Solving  
Problems in  
Subranges

### Data Structures II

- 1 Range Trees over Trees
- 2 Range Updates, Point Queries
- 3 Range Updates, Range Queries**
- Range Updates, Range Queries
- 4 Range Trees of Data Structures
- 5 Solving Problems in Subranges
- 6 Searching a Range Tree

Range Trees  
over Trees

Range  
Updates,  
Point  
Queries

Range  
Updates,  
Range  
Queries

Range  
Trees  
of Data  
Structures

Solving  
Problems in  
Subranges

Searching a  
Range Tree

- Now let's try range updates and range queries.

## Problem: Range Updates, Range Queries

29

## Range Updates, Range Queries

30

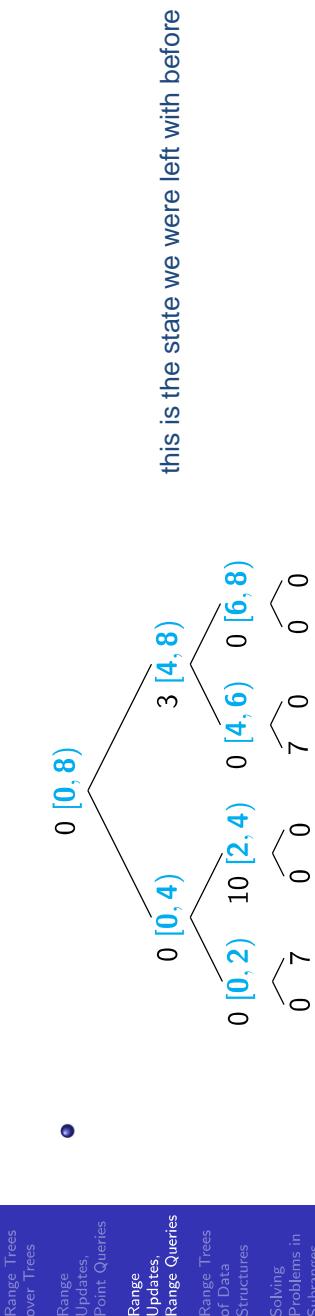
- We will support range updates in the same way we did for point queries. Instead, we will change how we do range queries.
    - In our earlier example, for each node we just stored the lazy counter. This was enough as every query involved walking from the root to a leaf.
    - However, recall to handle range queries in good time complexity we terminate our recursion once we've found a node that matches our current query range.
    - Hence for each node we will need to store 2 values, the previously we just stored the lazy value, now we also have to store the current result for this subtree (this is kind of like solutions to sub-problems in DP)
    - lazy counter and the sum of the node's range of responsibility.
    - 2 major changes:
      - ① Maintain for each node its lazy counter and the sum of its range.
      - ② Support updates through lazy propagation.
- so what is lazy propagation? Whenever you touch a node, propagate that nodes updates down to its children first.

- Lazy propagation is the idea that whenever we touch a node, we should propagate that node's updates to its children.
- For our example, propagate means add the lazy counter of node  $i$  to its two children and set the lazycounter of node  $i$  to 0.
- Essentially, instead of doing  $a[l, r] += v$ , we break the update into  $a[l, m] += v$  and  $a[m, r] += v$ .

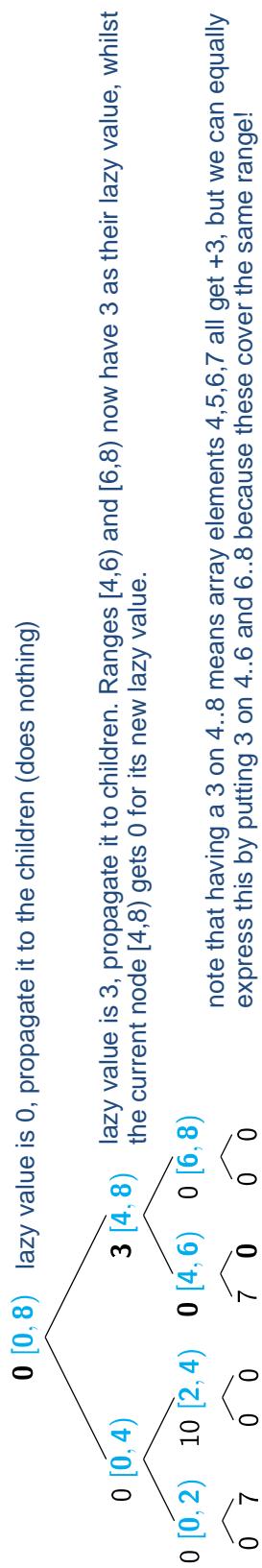
there is no need to "add the lazy value to the current node first", by propagating it down to the two children, that value is still be added to the whole range, you're just using two ranges to specify what was specified with one range originally. Remember that the parent of two ranges is equal to the union of those two ranges, so either adding the lazy value to a parent, or both of its children alternatively are really the same thing.

Lazy Propagation

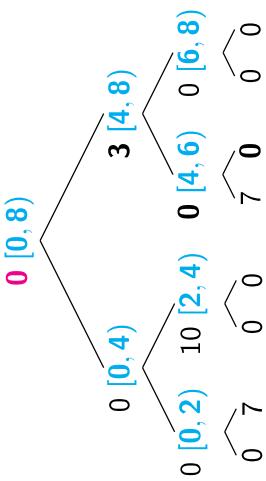
32



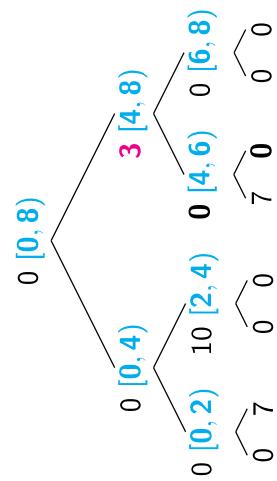
- Let's try querying  $a[5]$ .



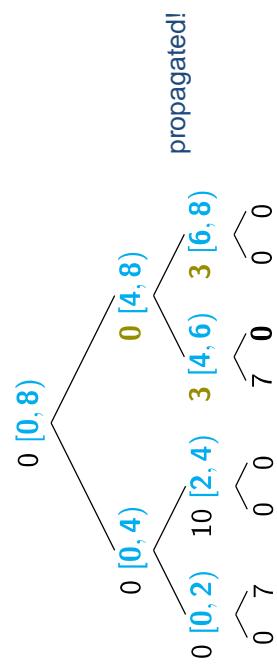
- Let's try querying  $a[5]$ .



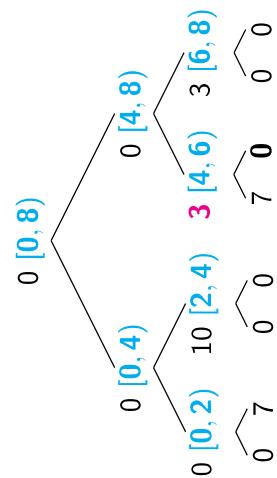
- Let's try querying  $a[5]$ .



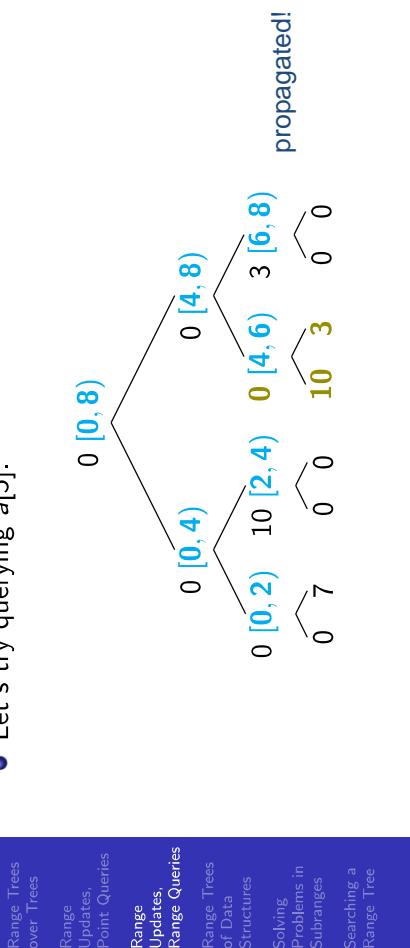
- Let's try querying  $a[5]$ .



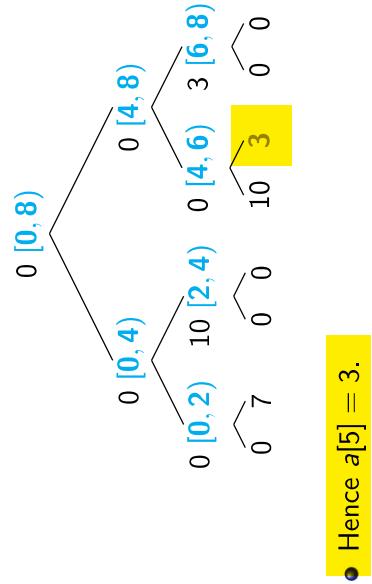
- Let's try querying  $a[5]$ .



- Let's try querying  $a[5]$ .



- Let's try querying  $a[5]$ .



- This ensures when we read the sum of a range from a node, we won't be missing any updates that are stored in the lazy counter of one of the node's ancestors.
- **Complexity Overhead?** No overhead, propagation is an  $O(1)$  operation per node.

## Storing Counter and Sum

41

### Data Structures II

#### Range Trees over Trees

#### Range Updates, Point Queries

#### Range Updates, Range Queries

#### Range of Data Structures

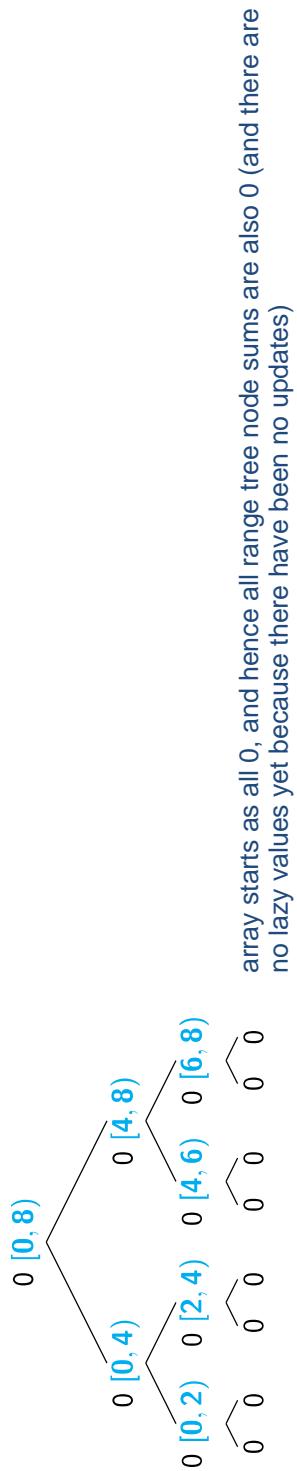
#### Solving Problems in Subranges

#### Searching a Range Tree

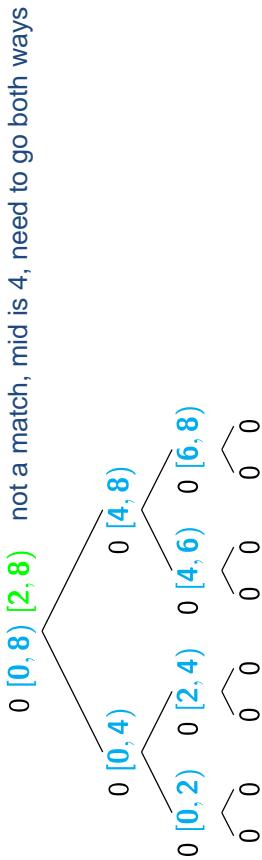
- To support range queries, for each node we also need to store the sum of its range.
- But because of range updates, we can't literally do this (else we'd need to update every node within the update range).
- Our invariant will be: Each node stores what the sum of its range would be, accounting only for lazy counters within its subtree.
- All lazy counters above each node are ignored.
- This way, an update only needs to modify the nodes encountered in the update's recursion.
- This will suffice since lazy propagation ensures when we actually query a node all its ancestors will have lazy counter 0.

because we have gotten here, it means that if there originally were lazy counters along the way, the would have been propagated all the way down to the leaf node (here), and thus they were all set to 0.

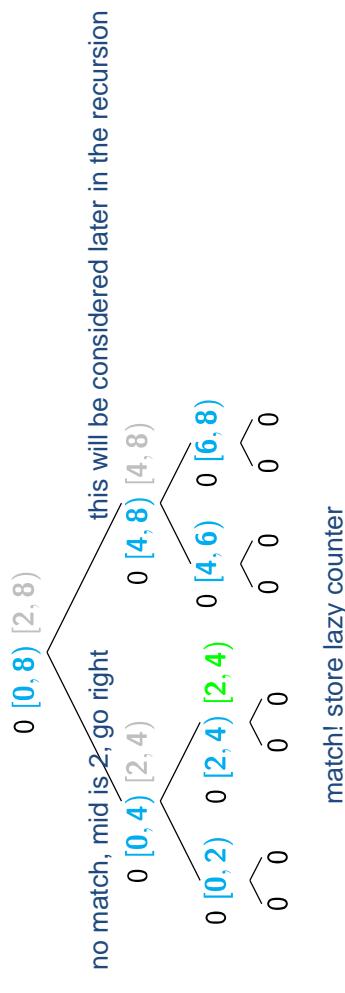
- Sum of a node's range will always be shown. Nonzero **lazy** counters will be written in brackets to the right.



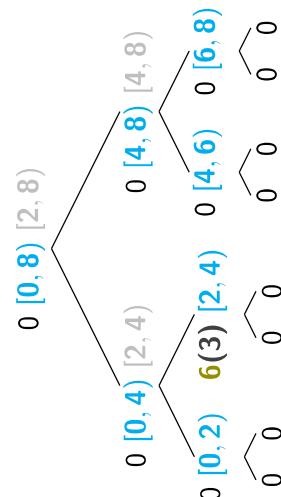
- Let's update the range  $[2, 8)$  with  $v = 3$ .



- Again this is done recursively. This is the same as for queries so I've shortcutted it.

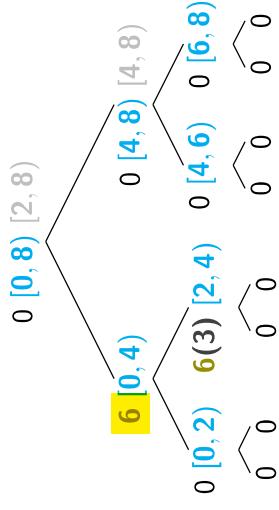


- Let's update the left side first. We need to update the lazy counter and also the sum.



the sum is 6 because this has 2 children in total, each of which needs 3 added to it, making the total 6.  
We keep the lazy counter here.

- We then return from this branch of the recursion. As we're returning we will update the nodes we passed through in this branch.



- Now our recursion enters the other branch. Same as before.

0 [0, 8) [2, 8) you can imagine there is a 6 here, waiting to be summed with the result of the right recursive branch.

Range  
Updates,  
Point Queries

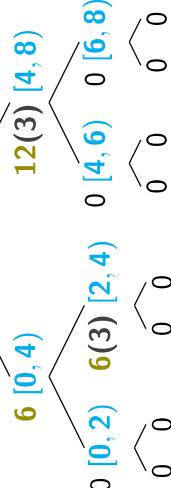
Range  
Trees

Range  
Queries

Range Trees  
of Data  
Structures

Solving  
Problems in  
Subranges

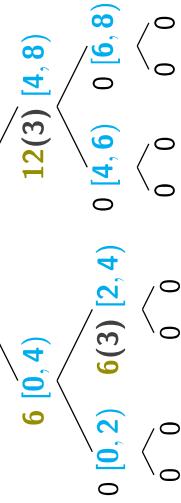
Searching a  
Range Tree



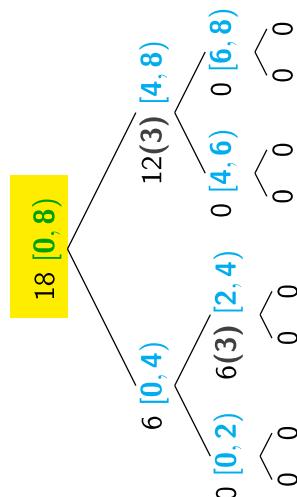
match! sum is 12 because there are 4 children in this subtree, all of which need 3 added to them.

- We now return from the right branch. We now update the root node before returning.

**18 [0, 8)** The 12 and the 6 make 18



- We now return from the right branch. We now update the root node before returning.



## Range Updates, Range Queries

50

Data

Structures II

Range Trees  
over Trees

Range  
Updates,

Point Queries

Range  
Updates

Range  
Queries

Range Trees  
of Data

Structures

Solving

Problems in

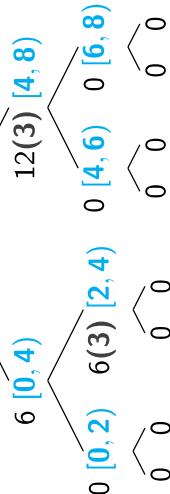
Subranges

Searching a

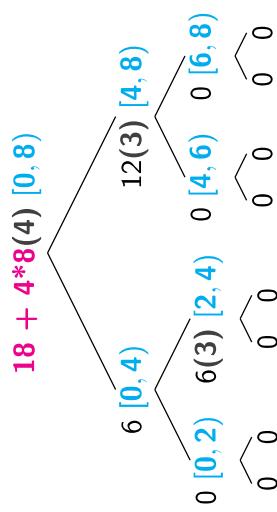
Range Tree

- Let's update a second update to the range  $[0, 8)$  with  $v = 4$ .

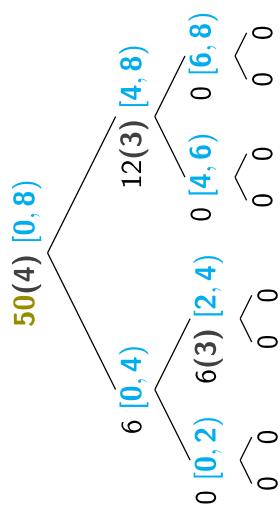
18  $[0, 8)$   $[0, 8)$  we get an instant match. There are 8 children, all of which need +4, hence sum += 32, and so sum = 50.  
We also get a lazy counter of 4.



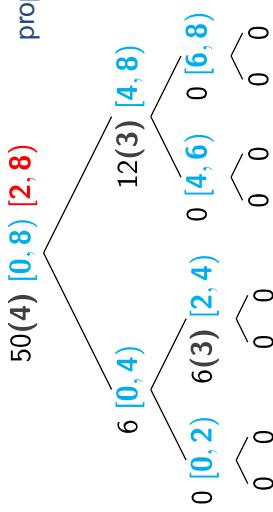
- Since the root's range matches we just update the root.



- Note how we did not modify any node but the root.



- Let's now query the sum of the range  $[2, 8)$ .



propagate the lazy counter down, meaning we get 4 on the left and 7 on the right

## Range Updates, Range Queries

54

Data

Structures II

Range Trees  
over Trees

Range

Updates,

Point Queries

Range

Updates,

Range Queries

Range Trees

of Data

Structures

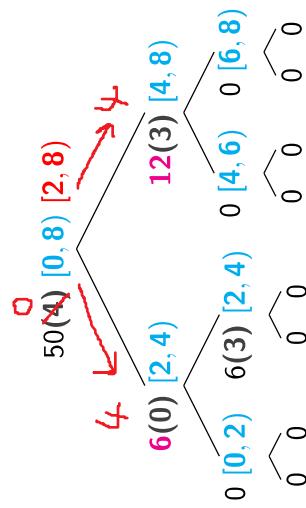
Solving

Problems in

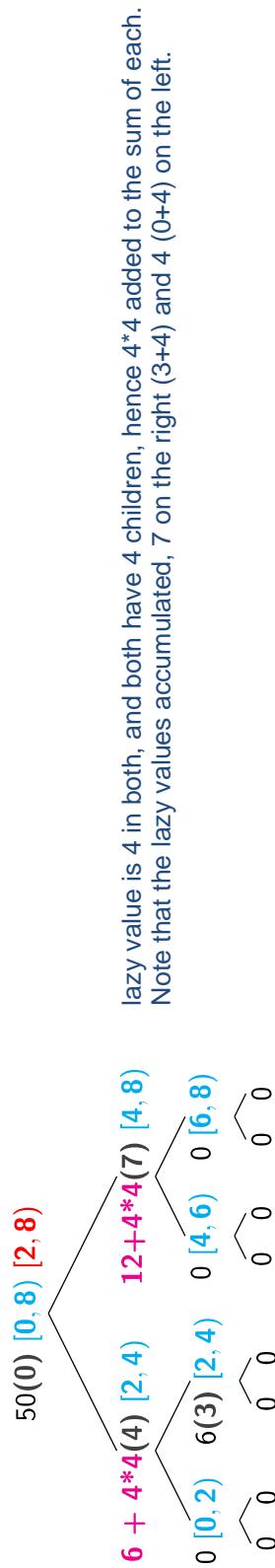
Subranges

Searching a  
Range Tree

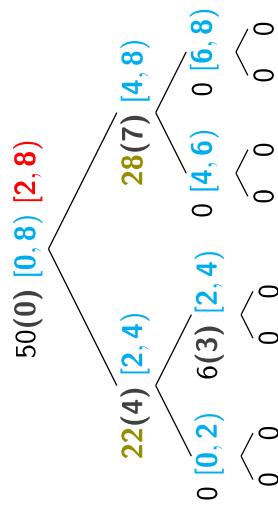
- Whenever we encounter a node, we lazy propagate out its lazy counter.



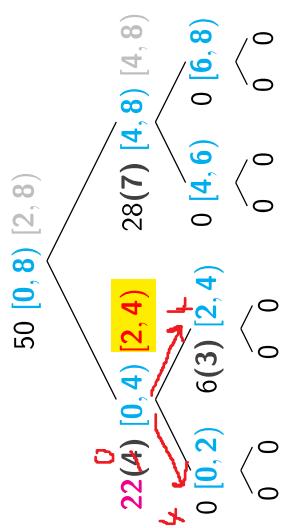
- When we lazy propagate, we also need to change node sums.



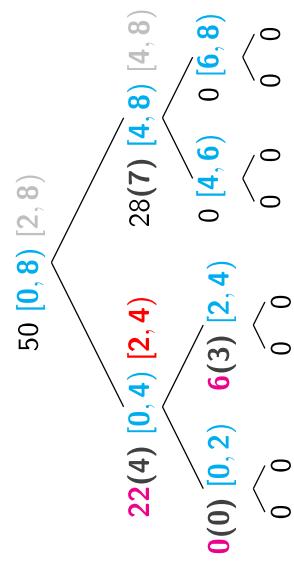
- When we lazy propagate, we also need to change node sums.



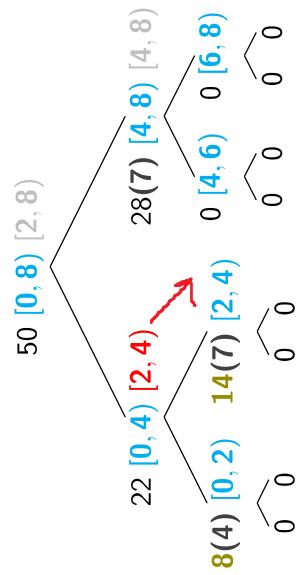
- Now we do the recursion for answering the query.



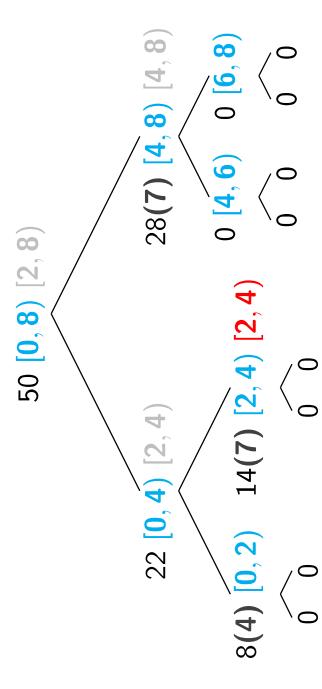
- Again, we need to lazy propagate.



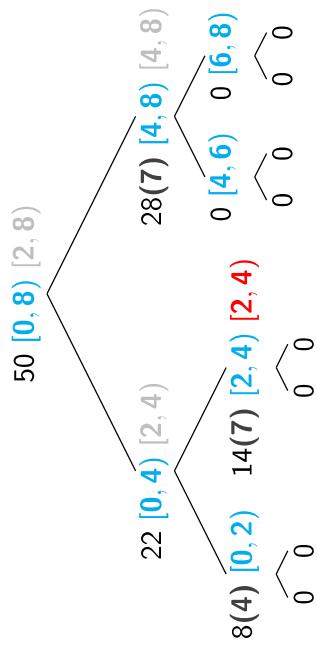
- Again, we need to lazy propagate.



- Now we recurse again.

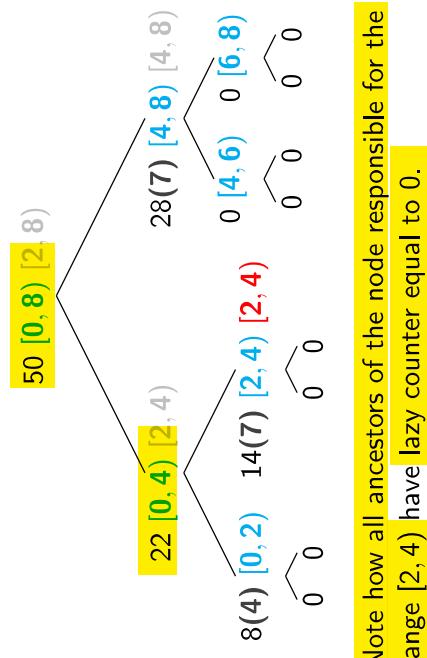


- For simplicity, we'll just say we **don't** lazy propagate when we've found the right range.



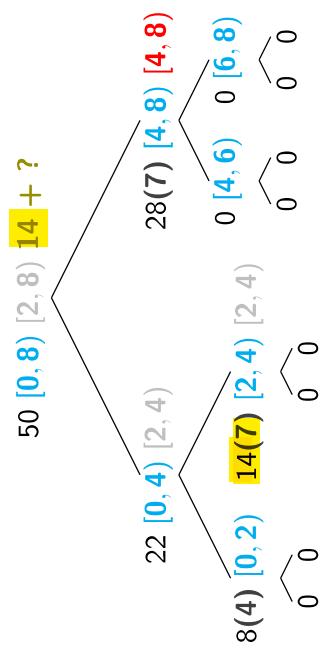
- So we **return the result we have obtained up the chain and continue the query in the other branch.**

- So we return the result we have obtained up the chain and continue the query in the other branch.



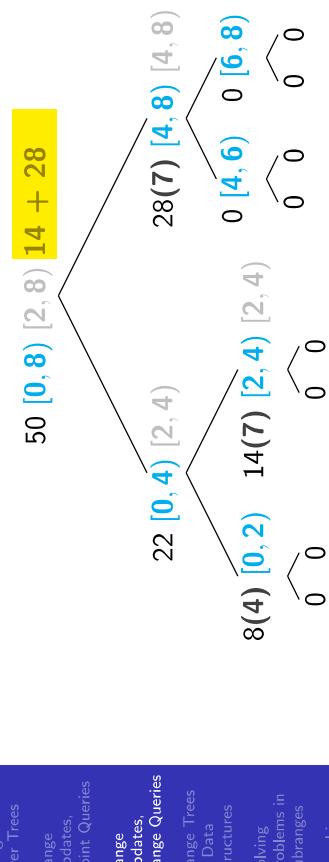
- Note how all ancestors of the node responsible for the range [2, 4) have lazy counter equal to 0.

- Now we continue in the second branch where we immediately find the node with the right range.



## Data Structures II

- So we immediately return with the value.



- And we now return from the root with the answer 42.

- Implementation wise, it helps to introduce some terminology.
  - In a recursion, call the **preorder** procedure the **procedure we call before recursing**.
  - Call the **postorder** procedure the procedure we call after we've returned from all children.
  - Then we will implement propagation as a **preorder** procedure.
  - And for updates, recalculating a node's sum is a **postorder** procedure.
- this makes sense, we wanted to propagate on visiting a node, which is in the pre-order step!
- this makes sense, you need to return the sums of the trees below to be able to calculate the sum of this tree.

Range Updates, Range Queries

## Range Updates, Range Queries

67

Data

```
// The root node is responsible for [0, MAX_N). Update range [uL, uR)
void update(int uL, int uR, int v, int i = 1, int cLeft = 0, int cRight = MAX_N) {
    if (uL == cLeft && uR == cRight) {
        updateLazy(i, v, cLeft, cRight);
        return;
    }
    propagate(i, cLeft, cRight); // pre-order propagation
    int mid = (cLeft + cRight) / 2;
    if (uL < mid) update(uL, min(uR, mid), v, i * 2, cLeft, mid);
    if (uR > mid) update(max(uL, mid), uR, v, i * 2 + 1, mid, cRight);
    recalculate(i, cLeft, cRight);
}

long long query(int qL, int qR, int i = 1, int cLeft = 0, int cRight = MAX_N) {
    if (qL == cLeft && qR == cRight) {
        return sum[i];
    }
    propagate(i, cLeft, cRight);
    int mid = (cLeft + cRight) / 2;
    long long ans = 0;
    if (qL < mid) ans += query(qL, min(qR, mid), i * 2, cLeft, mid);
    if (qR > mid) ans += query(max(qL, mid), qR, i * 2 + 1, mid, cRight);
    return ans;
}
```

Range Trees

over Trees

Range

Updates,

Point Queries

Range

Updates

Range

Queries

after all propagation and updates are done, you can calculate the new sums

- **Complexity:**  $O(\log n)$  per update/query still. We still visit the same nodes, the extra propagation and computation is just  $O(1)$  overhead per node.
- It is important to make sure you have invariants in mind when implementing range trees.
- For example, I had the invariant that `sum[i]` represents the sum accounting for all lazy updates in the subtree of  $i$ . Everything else was dictated by maintaining this invariant.
- You could instead have `sum[i]` account for all lazy updates in the subtree, excluding the lazy counter at node  $i$  itself.
- Doesn't matter, just stay consistent.

- **Problem Statement:** Given an array of integers  $A[N]$ , initially all 0, support  $M$  operations of the forms:
  - **Update**  $U \ a \ b \ v$ . Set  $A[a..b] = v$ ,  $v \geq 0$ .
  - **Query**  $Q \ a \ b$ . What is the max of  $A[a..b]$ ?
- **Input Format:** First line, 2 integers,  $N, M$ . The following  $M$  lines each describe an operation.
- **Constraints:**  $1 \leq N, M \leq 100,000$ .

## Example: Setting Ranges

70

### Sample Queries:

5 7	Q Q Q S S S Q
U 1 3 5	Q Q S S S S Q
U 2 4 1	Q Q S S S S S S
Q 1 3	Q Q S S S S S S
Q 2 3	Q S S S S S S S
U 3 4 3	Q S S S S S S S
Q 2 4	Q S S S S S S S
Q 1 10	Q S S S S S S S

### Sample Output:

5	5
1	1
3	3
5	5
-	-
3	3
5	5
1	1
1	1
5	5

## Example: Setting Ranges

71

- We will use the same lazy propagation framework.
- What our nodes store is dictated by the queries.
- Each of our nodes needs to store the max for their range of responsibility, *ignoring* all lazy values outside that node's subtree.
- Our lazy values are dictated by the updates.
- Each of our nodes needs to store the last update applied to the node.

- **Question?** For a given node in the range tree how do we know which update most recently covered the node's range?
- **Key Observation:** If we lazy propagate, it is the lazy value of the highest ancestor with a lazy value set.
- **Why?** Because whenever we apply an update, we **lazy** propagate existing updates on the path to the node we're updating. So no ancestors of the node have **lazy** values set. Hence the highest set **lazy** value is the most recent update.

## Example: Setting Ranges

73

- Now we know what we need.
- Our **lazy values** store the most recent update to a range.
- These will be **lazy propagated**. When we **lazy propagate** we just overwrite our children since we know our update is more recent than our children's.
- Each node stores the max of its range, based on only **lazy values** within its subtree.
- `maxrt[i] = lazy[i]` if `lazy[i]` is set, else it is the max of  $r$ 's children.

## Example: Setting Ranges

74

```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 100000;
const int UNSET = -1;
// Since A is initially, the default values are correct.
int lazyset[MAXN*3]; // UNSET if no lazy is set
int maxrt[MAXN*3];

// Recalculates a node's values assuming its children are correct.
// do NOT call these on leaves.
void recalculate(int i) {
    if (lazyset[i] != UNSET) maxrt[i] = lazyset[i];
    else maxrt[i] = max(maxrt[i*2], maxrt[i*2+1]);
}
void propagate(int i) {
    if (lazyset[i] == UNSET) return;
    lazyset[i*2] = lazyset[i*2+1] = lazyset[i];
    maxrt[i*2] = maxrt[i*2+1] = maxrt[i];
    lazyset[i] = UNSET;
}

Range Updates
Range Queries
Range Trees
over Trees
of Data
Structures
Solving
Problems in
Subranges
Searching a
Range Tree
```

## Example: Setting Ranges

75

```
Range Trees
over Trees
Range
Updates,
Point Queries
Range
Updates
Range Queries
Range Trees
of Data
Solving
Problems in
Subranges
Searching a
Range Tree
    void update(int uL, int vR, int v, int i=1, int cL=0, int cR=MAXN) {
        if (uL == cL && vR == cR) {
            layset[i] = maxrt[i] = v;
            return;
        }
        propagate(i);
        int mid = (cL + cR) / 2;
        if (uL < mid) update(uL, min(vR, mid), v, i*2, cL, mid);
        if (vR > mid) update(max(uL, mid), vR, v, i*2+1, mid, cR);
        recalculat(i);
    }

    int query(int qL, int qR, int i=1, int cL=0, int cR=MAXN) {
        if (qL == cL && qR == cR) {
            return maxrt[i];
        }
        propagate(i);
        int mid = (cL + cR) / 2;
        int ans = -1; // note all values are >= 0 in the question.
        if (qL < mid) ans = max(ans, query(qL, min(qR, mid), i*2, cL, mid));
        if (qR > mid) ans = max(ans, query(max(qL, mid), qR, i*2+1, mid, cR));
        return ans;
    }
```

## Data Structures II

- 1 Range Trees over Trees
  - Range Trees over Trees
  - Range Updates, Point Queries
- 2 Range Updates, Point Queries
- 3 Range Updates, Range Queries
  - Range Updates, Range Queries
- 4 Range Trees of Data Structures
  - Range Trees of Data Structures
  - Solving Problems in Subranges
- 5 Solving Problems in Subranges
  - Solving Problems in Subranges
- 6 Searching a Range Tree
  - Searching a Range Tree

- So far we've just used range trees to support operations on an array of integers.
- But the real power of range trees is in the way it decomposes ranges.
- The nodes can store anything.
- For example other data structures (!!)
- The most useful is probably a set or other SBBST.

- **Problem Statement** It's 2200 and climatology is now the most hectic job on earth. There is a constant deluge of rain predictions concerning the towns in LineLand. There are  $N$  towns in LineLand, in a line. Each prediction is of the form  $\cup a_i \ b_i \ d_i$  saying that there will be rain in towns  $[a_i, b_i)$  on day  $d_i$ . Interspersed among these updates, there will be queries of the form  $Q \ a_i \ d_i$ , asking if there is a predicted shower in town  $a_i$  on day  $d_i$ .
- **Input** First line,  $N, Q$ , the number of towns and operations.  $1 \leq N, Q \leq 100,000$ . Towns are 0 indexed. The next  $Q$  lines are the operations in the specified format.
- **Output** For each  $Q$  operation, 1 if there is forecasted rain and 0 otherwise.

• **Sample Input:**

```
10 6
U 0 3 1
Q 1 1
Q 1 2
Q 3 1
U 1 4 1
Q 3 1
```

• **Sample Output:**

```
1
0
0
1
```

- We have the characteristic range updates that suggest **range tree**.
- But it no longer suffices to store a single integer for each range.
- To store our predictions we should use a set.
- We will decompose the range of each prediction using the range tree and update the sets of each of the corresponding nodes.
- To answer a query, we need the predictions corresponding to each range containing the queried city. This is just the nodes on the path from the leaf to the root.

**Problem:** Should I bring an Umbrella?

8

- **Complexity?**  $O(N + Q \log^2 N)$ . Each update and query accesses  $O(\log N)$  nodes (this is a characteristic of the range decomposition itself) but each access costs  $O(\log N)$  due to the sets.
- **Warning:** We can't lazy propagate in this example. This is because the size of the data we are storing at each node isn't constant any more. So the cost of lazy propagation per operation is potentially  $O(N \log N)$  and this does not amortize.
- E.g.: have 50000 updates to the entire range, then have the next 50000 be queries forcing a  $O(N \log N)$  set copy each time.

- Range Trees over Trees
- Range Updates, Point Queries
- We can support other things sets and maps and OSTs support, like deleting predictions and finding the closest rain day or counting the number of cities raining on a given day in a range.
- **Moral:** If you need to store different kinds of data while supporting range operations, consider a range tree of a suitable data structure.

- Another classic problem, finding total area covered by a set of rectangles.

- **Problem Statement:** It's 2201 and you're done with Earth and its unpredictable rainfall. You've decided to move to Neptune. After landing, you find out, to your dismay, that not only does it rain on Neptune but it rains diamonds. But it's too late now to turn back so you'll just have to make do.  
As we all know, Neptune is a  $N \times N$  grid with bottom left corner  $(0, 0)$ . There are  $M$  diamond showers on Neptune, each a rectangle. You now wish to find how much of Neptune is covered by diamond showers.
- **Input:** First line 2 integers,  $N, M$ .  $1 \leq N, M \leq 100,000$ .
  - The next  $M$  lines are each of the form  $x_0\ y_0\ x_1\ y_1$ , describing a diamond shower with bottom left corner  $(x_0, y_0)$  and upper right corner  $(x_1, y_1)$ .
- **Output:** A single integer, the total area of Neptune covered by the union of all the showers.

## Problem: Mapping Neptune

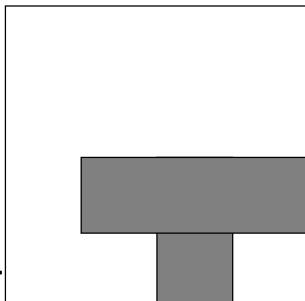
86

### Data Structures II

#### Sample Input:

4 2  
0 1 2 2  
1 0 2 3

#### Explanation:



#### Sample Output:

4

### Range Trees over Trees

### Range Updates, Point Queries

### Range Trees

### of Data Structures

### Solving Problems in Subranges

### Searching a Range Tree

#### Sample Output:

- 2 common approaches for 2D problems. Either a 2D data structure or a linear sweep in the y direction while maintaining a data structure over x.
- Latter is generally faster and easier.
- For each row, what do we need to track?
- Which columns currently have a rectangle.
- Standard way of doing this is create 2 events per rectangle, one at  $y_0$  instructing us to activate the rectangle, one at  $y_1$  instructing us to deactivate the rectangle.
- Suppose we have done this so we know which rectangles are active. How do we track how many columns have a rectangle?

- An active rectangle covers a *range* of x coordinates so range tree!
- The query we need to support is count the number of indices that are covered.
- We need to support the updates:
  - 1 Add a range.
  - 2 Remove a range.
- So we have a range update, range query situation.
- What do our nodes store and what are the lazy counters?

- What our nodes store is dictated by the queries.
- Each of our nodes needs to store the number of covered indices in its range.
- Our lazy counters are dictated by the updates.
- Each of our nodes needs to store whether a range fully covers that node's range.
- We can use a set for the lazy counter. Or we can use a counter.
- **Warning:** We can't `lazy` propagate here. Else deleting a range becomes a nuisance. (this becomes more natural if one thinks of the lazy counter as a set)

## Problem: Mapping Neptune

90

- So we decompose each update range same as how we always do.
- After decomposing the range, we update the **lazy counter** at the corresponding nodes.
- In addition, each node stores `freq[i]`, the number of covered indices in its range of responsibility **only** accounting for **lazy counters** in its subtree.
- Then `freq[i] = endRange[i] - startRange[i]` if `lazy[i] > 0`, otherwise, `freq[i]` is the sum of its two children.



## Problem: Mapping Neptune

92

```
Data Structures II
int N, M;
struct Event {
    int l, r, v;
    Event(int _l, int _r, int _v) : l(_l), r(_r), v(_v) {}
};

// Convention: process events for a y before calculating that value of y.
// When calculating y_i, we will count covered squares in [y_i, y_{i+1}]
vector<Event> events[1000005];

int main() {
    scanf("%d %d", &N, &M);
    for (int i = 0; i < M; i++) {
        int x0, y0, x1, y1;
        scanf("%d %d %d %d", &x0, &y0, &x1, &y1);
        events[y0].emplace_back(x0, x1, 1);
        events[y1].emplace_back(x0, x1, -1);
    }
    long long ans = 0;
    for (int i = 0; i < N; i++) {
        update(e.l, e.r, e.v);
        ans += query_total();
    }
    printf("%lld\n", ans);
    return 0;
}

Range Updates
Range Queries
Range Trees over Trees
Range Updates
Point Queries
Range Trees
of Data Structures
Solving Problems in Subranges
Searching a Range Tree
```

### Data Structures II

- 1 Range Trees over Trees
- 2 Range Updates, Point Queries
- 3 Range Updates, Range Queries
- 4 Range Trees of Data Structures
- 5 Solving Problems in Subranges
- 6 Searching a Range Tree

Range Trees  
over Trees

Range  
Updates,  
Point Queries

Range  
Updates  
Range Queries

Range Trees  
of Data  
Structures

Solving  
Problems in  
Subranges

Searching a  
Range Tree

- We can go further.
- By picking the right state to store we can solve many classic linear sweep problems except restricted to a subrange.
- Creating a range tree is kind of like applying divide and conquer in this view.

## Example: Maximum Sum Subrange

95

- **Problem Statement:** Given an array of integers  $A[N]$ , initially all 0, support  $M$  operations of the forms:
  - **Update**  $U \ i \ v$ . Set  $A[i] = v$ .
  - **Query**  $Q \ i \ j$ . Consider the sum of every (contiguous) subarray of  $A[i..j]$ . What's the maximum of these? Treat the empty subarray as having sum 0.
- **Input Format:** First line, 2 integers,  $N, M$ . The following  $M$  lines each describe an operation.
- **Constraints:**  $1 \leq N, M \leq 100,000$ .

## Example: Maximum Sum Subrange

96

### Sample Queries:

### Sample Output:

Range Updates, Point Queries	5 7	U 0 -2	0
Range Updates, Range Queries	U 2 -2	U 2 -2	3
Range Updates, Range Queries	U 1 3	U 1 3	4
Range Trees of Data Structures	Q 0 1	Q 0 1	
Solving Problems in Subranges	Q 0 100	Q 0 100	
Searching a Range Tree	U 3 3	U 3 3	
	Q 0 4	Q 0 4	

- The general strategy is to take a divide and conquer approach.
- Each node in our subtree stores the answer for queries that are exactly the node's range of responsibility.
- So we need to figure out how to merge 2 ranges so we can figure out the answer for a node from its two children and to answer queries.
- This is just like divide and conquer. The children nodes account for all answers that are contained within  $[l, m)$  or  $[m, r)$ .
- So the crucial (and difficult) part is accounting for possible solutions that cross  $m$ .

- For this, we will probably need to store additional metadata.
- Comes down to thinking about what a best solution crossing  $m$  must look like.
  - E.g: When we're trying to find the best subarray, for our left child, we generally want to store the best subarray of the form  $[x, m]$  and for our right child, we generally want to store the best subarray of the form  $[m, x]$ .
  - But remember, any metadata we add we also need to update in our range tree. However, generally this is easier because the metadata is more specific.

- Suppose we now know how to recalculate a node from its two children.
- Then answering a query should be easy.
- First break our query into subranges based on our range tree, as usual.
- Then use our recalculate procedure to merge these  $O(\log n)$  ranges.

- **Problem Statement:** Given an array of integers  $A[N]$ , initially all 0, support  $M$  operations of the forms:
  - **Update**  $U \ i \ v$ . Set  $A[i] = v$ .
  - **Query**  $Q \ i \ j$ . Consider the sum of every (contiguous) subarray of  $A[i..j]$ . What's the maximum of these? Treat the empty subarray as having sum 0.
- **Input Format:** First line, 2 integers,  $N, M$ . The following  $M$  lines each describe an operation.
- **Constraints:**  $1 \leq N, M \leq 100,000$ .

**Sample Queries:**

Range Trees  
over Trees  
Range  
Updates,  
Point Queries  
Range  
Updates  
Range Queries  
Range Trees  
of Data  
Structures  
Solving  
Problems in  
Subranges  
Searching a  
Range Tree

**Sample Output:**

5 7  
U 0 -2  
U 2 -2  
U 1 3  
Q 0 1  
Q 0 100  
U 3 3  
Q 0 4

- Our end goal is a range tree where each node stores the best answer for its range of responsibility.
- The difficult part is merging two nodes.
- Let's say we have a node responsible for the range  $[l, r]$  with children responsible for the ranges  $[l, m)$  and  $[m, r)$ .
- If the best subarray is solely in  $[l, m)$  or solely in  $[m, r)$  then we are done. What can we say about subarrays crossing  $m$ ?
- **Observation:** They should start at  $st$  such that  $[st, m)$  has maximum possible sum. They should similarly end at an  $en$  such that  $[m, en)$  has maximum possible sum.

- Range Trees over Trees
  - Range Updates, Point Queries
  - Range Updates Range Queries
  - Solving Problems in Subranges
  - Searching a Range Tree
- So for each node we should store the maximum possible sum of a subarray of the form  $[l, x)$  and of the form  $[x, r)$ .
  - Call this `bestStart[i]` and `bestEnd[i]`.
  - But now we have the same problem. How do we update `bestStart[i]` and `bestEnd[i]` from the 2 children of  $i$ ?
  - Again, we follow the same approach.
  - If `bestStart[i]` is from a subarray contained entirely in the left child then we are done.
  - Otherwise, what can it look like?
  - **Observation:** It is of the form  $[l, m) \cup [m, x)$  where  $x$  corresponds to `bestStart[rightChild]`.

- So
  - bestStart[i] = max(bestStart[leftChild], sum[leftChild] + bestStart[rightChild])

where  $\text{sum}[i]$  is the sum of  $i$ 's entire range.

  - So we now need to maintain  $\text{sum}[i]$ .
  - But this is easy, you've seen this many times.
  - Phew! We're done now. Only needed to go 3 levels deep!

## Example: Maximum Sum Subrange

105

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100000;
struct state {
    long long bestStart, bestEnd, sum, bestSubarray;
};

Range Trees
over Trees
Range
Updates,
Point Queries
Range
Updates
Range Queries
Range Trees
of Data
Structures II
{
    state mergeStates(state& left, const state& right) {
        state ret;
        ret.bestStart = max(left.bestStart, left.sum + right.bestStart);
        ret.bestEnd = max(right.bestEnd, left.bestEnd + right.sum);
        ret.sum = left.sum + right.sum;
        ret.bestSubarray = max(max(left.bestSubarray, right.bestSubarray),
            left.bestEnd + right.bestStart);
        return ret;
    }

    // Default value of state is all 0. This is correct for us.
    state rt[MAXN+3];
};

void update(int p, int v, int i=1, int cL=0, int cR=MAXN) {
    if (cR - cL == 1) {
        rt[i].sum = v;
        rt[i].bestStart = rt[i].bestEnd = rt[i].bestSubarray = max(v, 0);
        return;
    }
    int mid = (cL + cR) / 2;
    if (p < mid) update(p, v, i * 2, cL, mid);
    else update(p, v, i * 2 + 1, mid, cR);
    rt[i] = mergeStates(rt[i*2], rt[i*2+1]);
}

Solving
Problems in
Subranges
Searching a
Range Tree
```

```

Range Updates, Point Queries
Range Updates, Range Queries
Range of Data Structures
    state query(int qL, int qR, int i=1, int cL=0, int cR=MAXN) {
        if (qL == cL && qR == cR) {
            return rt[i];
        }
        int mid = (cL + cR) / 2;
        if (qR <= mid) return query(qL, qR, i * 2, cL, mid);
        if (qL >= mid) return query(qL, qR, i * 2 + 1, mid, cR);
        return mergeStates(
            query(qL, min(qR, mid), i * 2, cL, mid),
            query(max(qL, mid), qR, i * 2 + 1, mid, cR));
    }
}

```

- **Complexity?** Still  $O(\log N)$  for everything, `mergeStates` is an  $O(1)$  operation.
- **Moral:** While the solution seems involved, the general strategy is very simple. Repeatedly consider what is needed to merge 2 different states and see what additional metadata is necessary. Then hope this stabilizes.

- We can apply this for many simple problems on a line.
- We can also apply this to some DP problems that have small state space at any point.
- For these, your nodes store matrices detailing how to transition between states.

**Data  
Structures II**

- 1 Range Trees over Trees
- 2 Range Updates, Point Queries
- 3 Range Updates, Range Queries
- 4 Range Trees of Data Structures
- 5 Solving Problems in Subranges
- 6 **Searching a Range Tree**

Range Trees  
over TreesRange  
Updates,  
Point QueriesRange  
Updates  
Range QueriesRange Trees  
of Data  
StructuresSolving  
Problems in  
SubrangesSearching a  
Range Tree

- For most data structures it suffices to treat them as a black box.
- Hopefully by now you've gotten the sense that this is less true for range trees.
- Sometimes it is useful to also modify how we traverse a range tree.
- This is mainly useful when we are searching for the first/any value that satisfies some given constraint.

- Let's say we want to find any value that satisfies a criterion  $X$ .
- For concreteness, let's say we want to find any value that's at least  $L$ .
- In each node, we store enough data to determine if there is a value in its range that satisfies  $X$ .
- For our example, we can store the max of all values in each range.
- Once we have this, finding a value is easy. We know for both children whether there is a value inside their range that satisfies  $X$ . We then just recurse into whichever side has a value that satisfies  $X$ .
- To find the leftmost/rightmost such value, we just bias our search towards the left or right child.

- Suppose now we want to find if any value in a given range  $[l, r)$  that satisfies criterion  $X$ .
- Now we just decompose  $[l, r)$  into  $O(\log n)$  ranges as we usually do with a range tree.
- We can then just repeat this for each of the nodes in our decomposition.
- **Complexity?**  $O(\log n)$  if you implement correctly since we actually only need to do this once, to the first node which we know contains a value satisfying  $X$ .
  - Again, easy to find leftmost/rightmost.

- **Problem Statement:** Given an array,  $A[N]$ , all initially 0.  
Support  $M$  operations of the forms:
  - **Update**  $U$   $a$   $v$ . Set  $A[a] = v$ .
  - **Query**  $Q$   $a$   $b$   $v$ . What's the minimum index  $a \leq i < b$  such that  $A[i] > v$ , or -1 if no such index exists.
- **Input Format:** First line,  $N, M$ . Next  $M$  lines describe the operations.
- **Constraints:**  $1 \leq N, M \leq 100,000$ .

4	7		-1
U	0	2	
U	1	3	
Q	0	4	2
Q	0	4	3
Q	0	4	1
U	0	4	
Q	0	4	2

- To guide our search we need to know whether a range contains a value that is at least  $v$ .
- For this, it suffices to store the max of each range.
- We know how to maintain this, it's just a point update, range query range tree.
- Now to find a value that is at least  $v$  we just need to search only nodes with  $\max[i] > v$  and terminate our search once we have found a value.
- To find the first such  $i$ , just always search the left child's subtree first.

### Implementation Details:

- So far we've only recursed into nodes we need to by checking before recursing. For this it is a bit easier to always recurse and return immediately if we've recursed into a node whose range is disjoint from the query range.
- To find an index we have to recurse down to the leaves.
  - So we no longer early exit when the query range is the same as the node's range.
- Instead we early terminate once we have found a leaf. To support this, our recursion will return a boolean indicating if we have found an index.

```

Range Trees
over Trees
using namespace std;
const int MAXN = 100000;
int maxrt[MAXN*3];

// Standard max range tree.
void update(int p, int v, int i=1, int cL=0, int cR=MAXN) {
    if (cR - cL == 1) {
        maxrt[i] = v;
        return;
    }
    int mid = (cL + cR) / 2;
    if (p < mid) update(p, v, i*2, cL, mid);
    else update(p, v, i*2+1, mid, cR);
    maxrt[i] = max(maxrt[i*2], maxrt[i*2+1]);
}

Solving
Problems in
Subranges
Searching a
Range Tree

```

```
#include <bits/stdc++.h>
```

```
using
```

```
namespace
```

```
std;
```

```
const int MAXN = 100000;
```

```
int maxrt[MAXN*3];
```

```
// Standard max range tree.
```

```
void update(int p, int v, int i=1, int cL=0, int cR=MAXN) {
```

```
    if (cR - cL == 1) {
```

```
        maxrt[i] = v;
```

```
        return;
```

```
    }
```

```
    int mid = (cL + cR) / 2;
```

```
    if (p < mid) update(p, v, i*2, cL, mid);
```

```
    else update(p, v, i*2+1, mid, cR);
```

```
    maxrt[i] = max(maxrt[i*2], maxrt[i*2+1]);
```

```
}
```

```

Range Trees
    Range Queries
        Range Trees
            Range Updates
                Point Queries
                    Range Updates
                        Range Queries
                            Range of Data Structures
                                Solving Problems in Subranges
                                    Searching a Range Tree
}
| bool query_rec(int qL, int qR, int v, int &foundP1c,
|     int i1, int cL=0, int cR=MAXN) {
|     // Query range does not intersect the node's range.
|     if (qL >= cR || qR <= cL) return false;
|     // Nothing in i's range is big enough
|     if (maxxt[i] <= v) return false;
|     if (cR - cL == 1) {
|         foundP1c = cL;
|         return true;
|     }
|     int mid = (cL + cR) / 2;
|     if (query_rec(qL, qR, v, foundP1c, i*2, cL, mid)) return true;
|     if (query_rec(qL, qR, v, foundP1c, i*2+1, mid, cR)) return true;
|     return false;
| }
| int query(int qL, int qR, int v) {
|     int ans = -1;
|     query_rec(qL, qR, v, ans);
|     return ans;
| }
}

```

- **Complexity?** Actually still  $O(\log n)$  per operation.
- Recall our previous recursions stopped whenever we encountered a node whose range was entirely contained in  $[qL, qR]$ .
- In this recursion, whenever we encounter such a node, either its max value is too low and we stop anyways, or the node contains the index we are looking for.
- The latter case only occurs once and the search for the index is  $O(\log n)$  since we only recurse from a node if we know for sure its range contains the desired index.

- This trick is useful for finding if an event has occurred in the array.

**Problem Statement:**

- It is 2155 and Earth has been renamed Water. LineLand with its constant showers has been particularly devastated. LineLand consists of  $N$  towns in a row, each with a height  $h_i$ . Initially all of these have water level 0.
- The climatologists of LineLand forecast there will be  $M$  showers, the  $i$ -th raising the water levels of towns  $[a_i, b_i)$  by  $w_i$ .

The mayor of LineLand wants to know how many towns are underwater (total water level is greater than the height of the town) after each shower.

**Input Format:** First line, 2 integers,  $N, M$ .

- $1 \leq N, M \leq 500,000$ . Next line,  $N$  integers, the initial heights of the towns. Next  $M$  lines each describe a shower.

**Sample Input:**

3 2  
1 4 2  
0 2 3  
1 3 2

**Sample Output:**

1  
2

- **Observation 1:** Once a town is underwater, it is always underwater.
  - So we just need to find out what towns change from above water to underwater after each operation.
  - What is the criterion for a town to be underwater?
  - That `total_water[i] > height[i]`.
  - Alternatively that  $0 > \text{height}[i] - \text{total_water}[i]$ .

- So to know if there is a new town underwater, we just need to know if  $\min(\text{height}[i] - \text{total_water}[i]) < 0$ .
- We can then delete this town so we do not count it more than once then repeat.
- For this problem, setting a town's height to infinity is as good as deleting the town.

```

Data Structures II

Range Trees over Trees
Range Updates, Point Queries
Range Updates, Range Queries
Solving Problems in Subranges
Searching a Range Tree

#include <bits/stdc++.h>
using namespace std;

const int MAXN = 500000;
// Make this large enough that it will never go underwater.
const int INF = 1e9;
// Standard range update min range tree.
int minr(MAXN*3);
// Standard function for A[uL, uR) += v
void update(int uL, int uR, int v);
// Query that returns index of anything with value < 0
// Or -1 if no such value exists.
int query(int qL, int qR, int v);

int main() {
    cin >> N >> M;
    for (int i = 0; i < N; i++) {
        int ch; cin >> ch; update(i, i+1, ch);
    }
    int ans = 0;
    for (int i = 0; i < N; i++) {
        int a, b, w, cInd; cin >> a >> b >> w;
        update(a, b, -w);
        while ((cInd = query(0, MAXN, 0)) != -1) {
            ans++;
            update(cInd, cInd+1, INF); // "delete" cInd
        }
        cout << ans << endl;
    }
}

```