Data Structures II

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Sydney

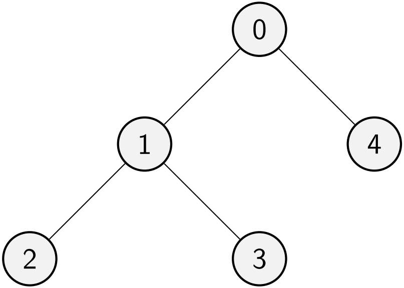
Term 3, 2023

Table of Contents

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

Range Tree on Trees

- **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 $U\ i\ w$. Change the value of node i to i.
  - **Query:** Of the form $Q\ i$. What is the sum of values in the subtree rooted at vertex i?
- **Input** First line, n, q, number of vertices and operations. $1 \leq n, q \leq 100,000$. The next line specifies the tree. $n - 1$ integers, p_i, the parent of vertex i (1-indexed).
  The following q lines describe the updates and queries. $1 \leq n, q \leq 100,000$.
- **Output** For each **Query**, an integer, the sum of values in the subtree rooted at vertex i.



Sample Input:

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

Sample Output:

```
3
1
4
```

Range Tree on Trees

- 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.
- 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.

Range Tree on Trees

- **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.

Range Tree on Trees


```
#include <vector>
using namespace std;

const int N = 100100;
// Suppose you already have your tree set up.
vector<int> children[N];
// A node is responsible for the range [startRange[v], endRange[v])
int indexInRangeTree[N], startRange[N], endRange[N];
int totId;
// A range tree that supports point update, range sum queries.
long long rangeTree[1<<18];
void update(int plc, int v);
long long query(int qL, int qR); // Query for [qL, qR)

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

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

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

Table of Contents

Range Updates, Point 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.

Problem: Range Updates, Point Queries

- Given an array $a[n]$, initially all zeros, support q operations, each being one of the following forms:
  - **Update:** `U l r v`. Perform `a[l,r) += v`.
  - **Query:** `Q x`. Output `a[x]`.
- $n, q \leq 100,000$.

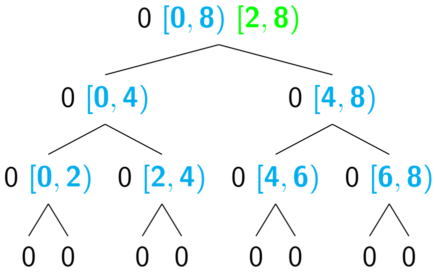
Range Updates, Point Queries

- 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[l,r) += x`.
- This is the same as performing `a[l,m) += x` and `a[m,r) += x`.
- So we can partition our initial update into smaller ranges however we wish.

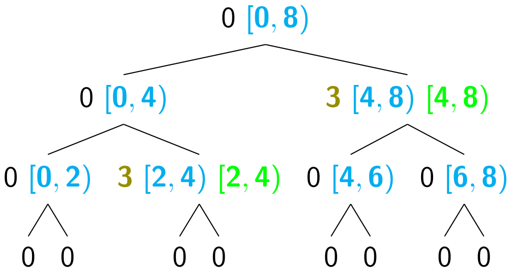
Range Updates, Point Queries

- 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.

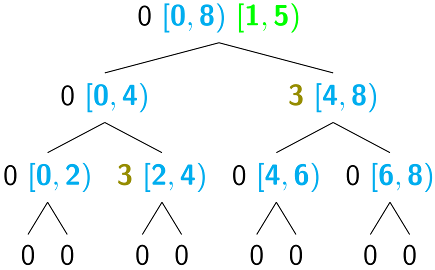
- Let's update the range $[2,8)$ with $v = 3$.



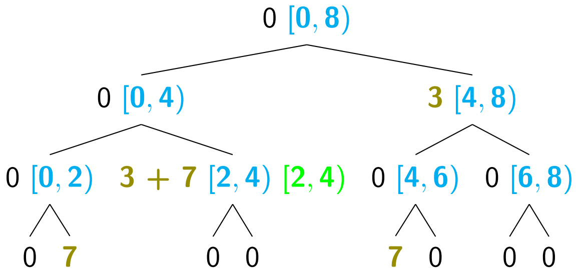
- As with range queries, we will push the update range down into the applicable nodes.



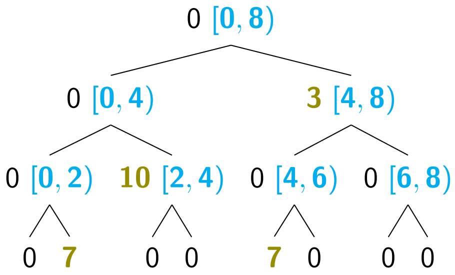
- Now let's assume we've been given a second update, for the range $[1,5)$ with $v = 7$.



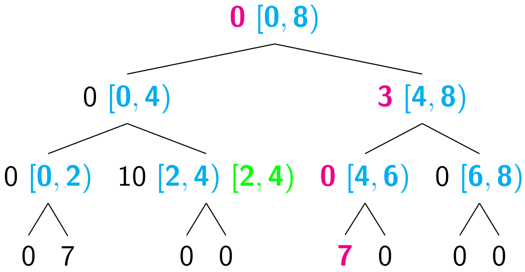
- The range decomposition is $[1, 2), [2, 4), [4, 5)$.



- 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 $a[4] = 10$.

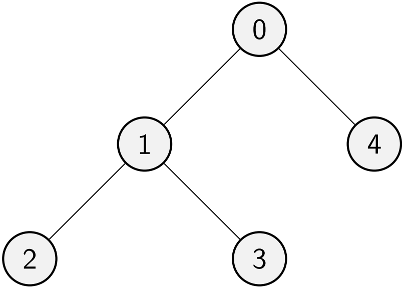
```
const int N = 100100;
long long lazyadd[1<<18];

int n;

// The root node is responsible for [0, n). Update range [uL, uR)
// Compare to range query code.
void update(int uL, int uR, int v, int i = 1, int cL = 0, int cR = n) {
  if (uL == cL && uR == cR) {
    lazyadd[i] += v;
    return;
  }
  int mid = (cL + cR) / 2;
  if (uL < mid) update(uL, min(uR, mid), v, i * 2, cL, mid);
  if (uR > mid) update(max(uL, mid), uR, v, i * 2 + 1, mid, cR);
}

long long query(int p, int i = 1, int cL = 0, int cR = n) {
  if (cR - cL == 1) {
    return lazyadd[i];
  }
  int mid = (cL + cR) / 2;
  long long ans = lazyadd[i];
  if (p < mid) ans += query(p, i * 2, cL, mid);
  else ans += query(p, i * 2 + 1, mid, cR);
  return ans;
}
```

Range Updates, Point Queries

- **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 $a[i] = a[i] \mod v_q$ 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 finnicky and depends on the specific updates and probably requires lazy propagation.

Dynamic Distance on a Tree

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



Sample Input:

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

Sample Output:

```
3
7
```

Dynamic Distance on a Tree

- Recall our solution for the static problem.
- Let $l = \text{lca}(i, j)$. We split the path from i to j into a path from i to l followed by a path from j to l.
- Let `weight_sum(i)` be the sum of weights from the root to i. The answer is then just `weight_sum(i) + weight_sum(j) - 2*weight_sum(l)`.
- Our updates don't change the tree structure but change `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.

```
using namespace std;

const int N = 100100;

// Suppose you already have your tree set up.
int depth[N]; // Depth in tree (ignores weight).
int lca(int a, int b);

// A node is responsible for the range [startRange[v], endRange[v])
int indexInRangeTree[N], startRange[N], endRange[N];
// A range tree supporting range updates of add, point queries of value.
long long rangeTree[1<<18];
void update(int uL, int uR, long long v); // value[uL,uR) += v
long long query(int q);

void update_edge(int i, int j, long long v) {
    // To update the edge's subtree, we need to know which of the 2 nodes are
        lower.
    if (depth[i] > depth[j])
        swap(i, j);
    update(startRange[j], endRange[j], v);
}

long long get_tree_distance(int i, int j) {
    int l = lca(i, j);
    return query(indexInRangeTree[i]) + query(indexInRangeTree[j])
        - 2*query(indexInRangeTree[l]);
}
```

Table of Contents

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

Problem: Range Updates, Range Queries

- Given an array $a[n]$, initially all zeros, support q operations, each being one of the following forms:
  - **Update:** `U l r v`. Perform `a[l,r) += v`.
  - **Query:** `Q l r`. Output $\sum_{i=l}^{r-1} a[i]$.
- $n, q \leq 100,000$.

Range Updates, Range Queries

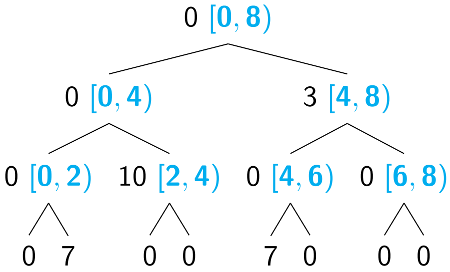
- We will support range updates in the same way we did for point queries. Instead, we will change how we do range queries.
- In our earlier example, for each node we just stored the lazy counter. This was enough as every query involved walking from the root to a leaf.
- However, recall to handle range 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 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*.

Lazy Propagation

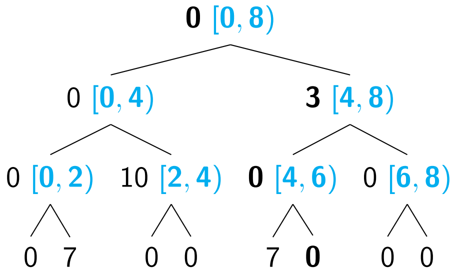
- *Lazy propagation* is the idea that whenever we touch a node, we should propagate that node's updates to its children.

- 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`.

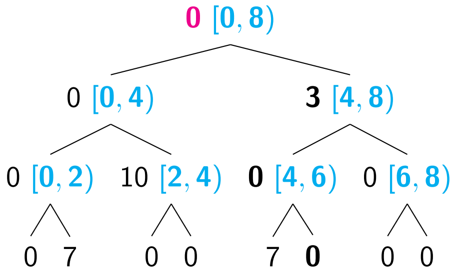

0 [0, 8)
0 [0, 4)
3 [4, 8)
0 [0, 2)
10 [2, 4)
0 [4, 6)
0 [6, 8)
0 7
0 0
7 0
0 0

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

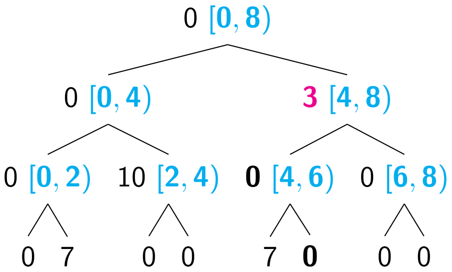


Lazy Propagation

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

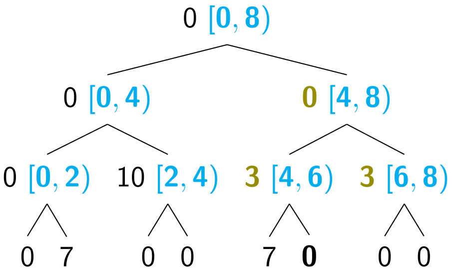


- Let's try querying $a[5]$.

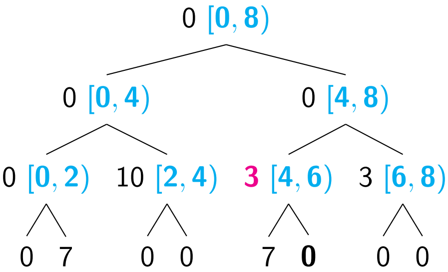


Lazy Propagation

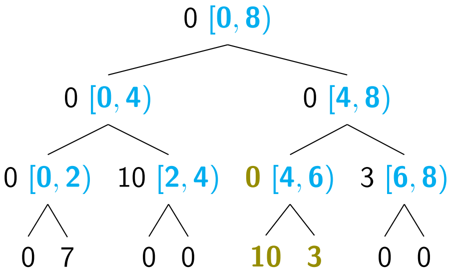
- Let's try querying $a[5]$.



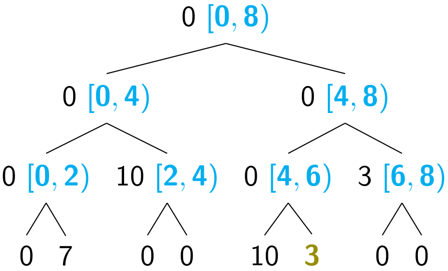
- Let's try querying $a[5]$.



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



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



- Hence $a[5] = 3$.

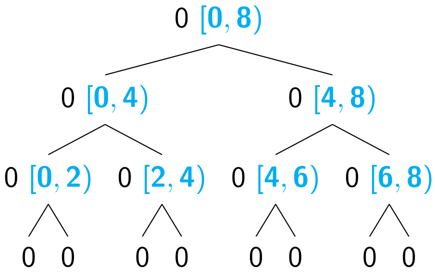
- This ensures when we read the sum of a range from a node, we won't be missing any updates that are stored in the lazy counter of one of the node's ancestors.

- **Complexity Overhead?** No overhead, propagation is an $O(1)$ operation per node.

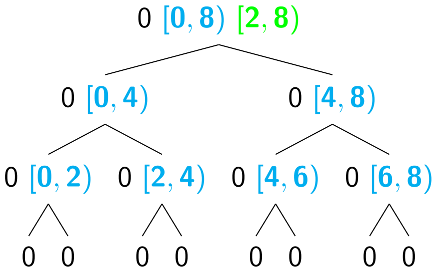
Storing Counter and Sum

- 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.

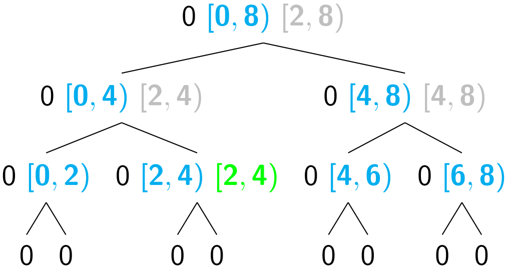
- Sum of a node's range will always be shown.
- Nonzero lazy counters will be written in brackets to the right.



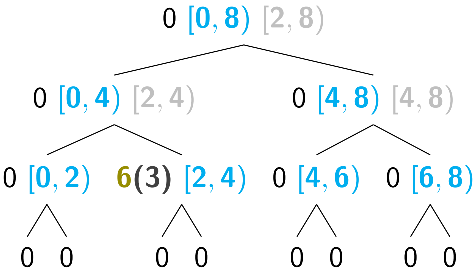
- Let's update the range $[2, 8)$ with $v = 3$.



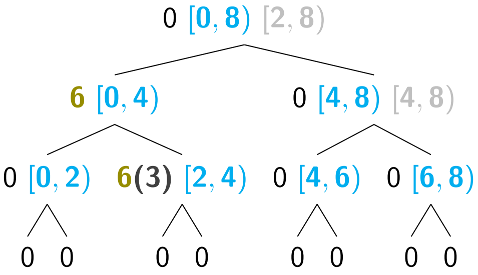
- This is done recursively, just like queries, so we'll summarise.



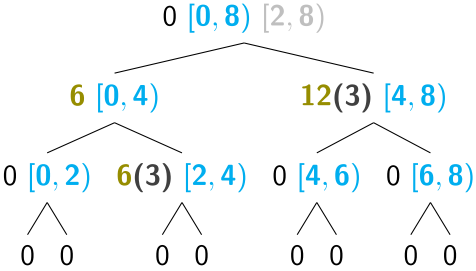
- Let's update the left side first.
- We need to update the lazy counter and also the sum.



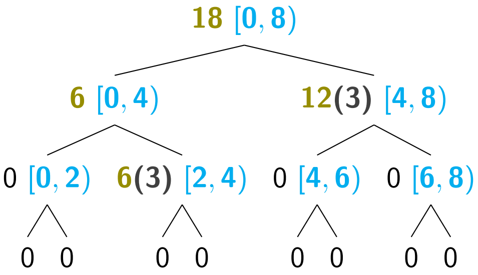
- We then return from this branch of the recursion.
- As we're returning we will update the nodes we passed through in this branch.



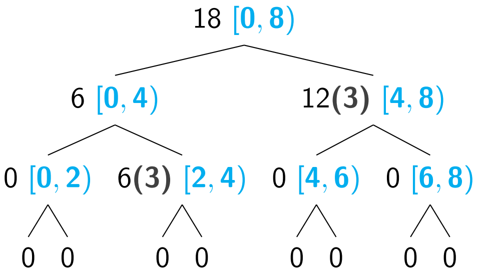
- Now our recursion enters the other branch. Same as before.



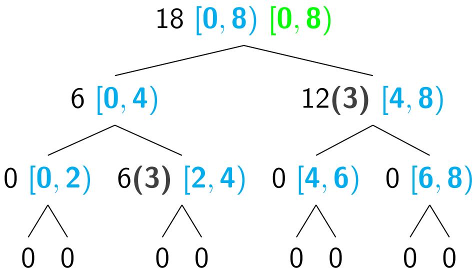
- We now return from the right branch.
- We now update the root node before returning.



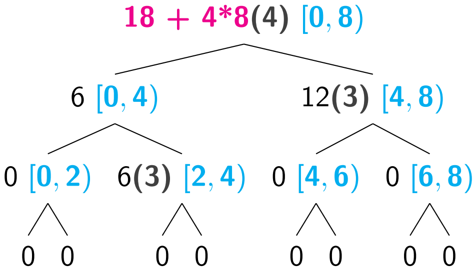
- We now return from the right branch.
- We now update the root node before returning.



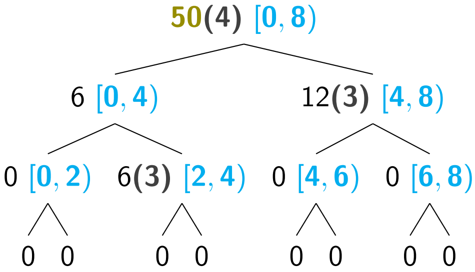
- Let's update a second update to the range $[0, 8)$ with $v = 4$.



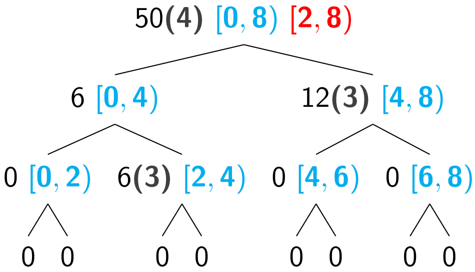
- Since the root's range matches we just update the root.



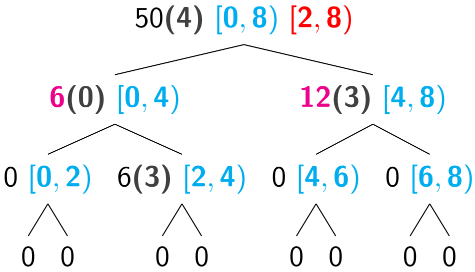
- Note how we did not modify any node but the root.



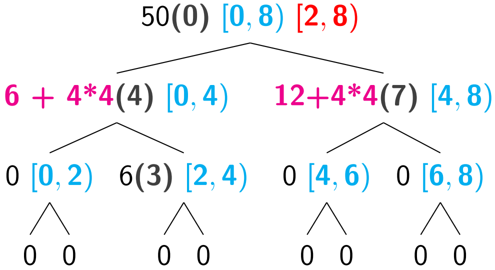
- Let's now query the sum of the range $[2, 8)$.



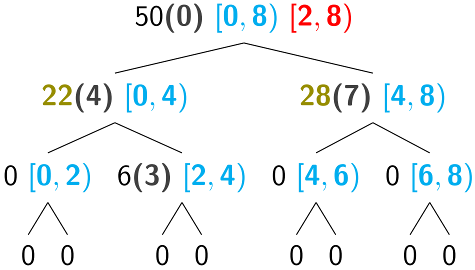
- Whenever we encounter a node, we lazy propagate out its lazy counter.



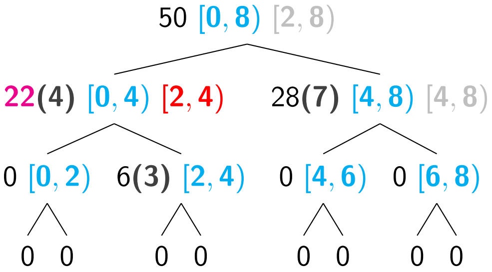
- When we lazy propagate, we also need to change node sums.



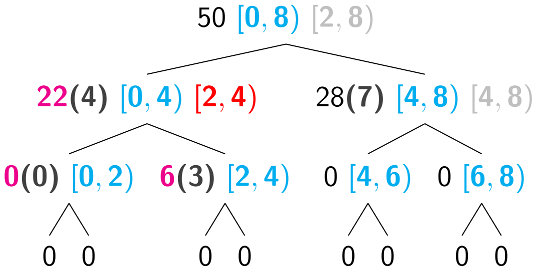
- When we lazy propagate, we also need to change node sums.



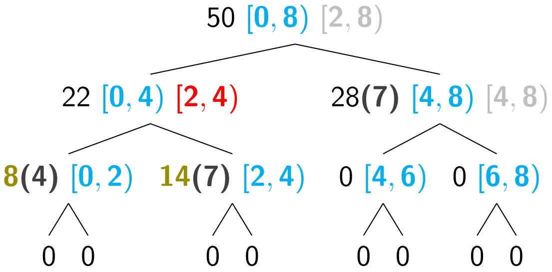
- Now we do the recursion for answering the query.



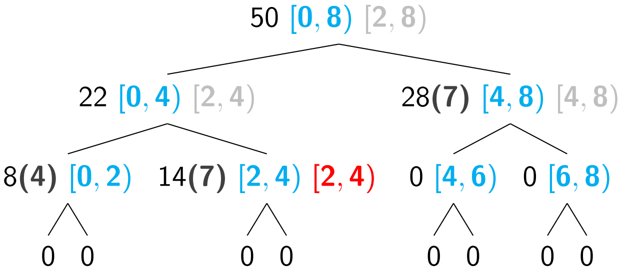
- Again, we need to lazy propagate.



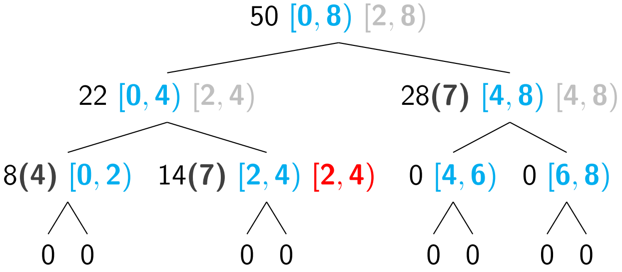
- Again, we need to lazy propagate.



- Now we recurse again.

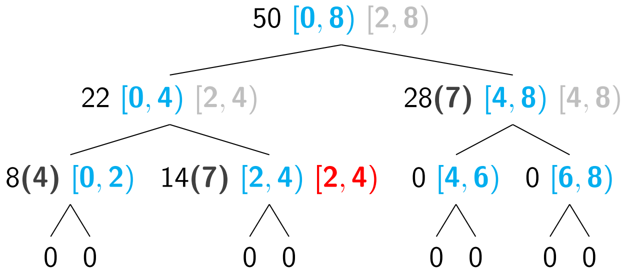


- For simplicity, we'll just say we don't lazy propagate when we've found the right range.



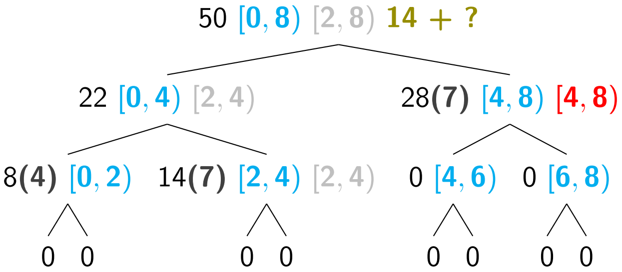
- So we return the result we have obtained up the chain and continue the query in the other branch.

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

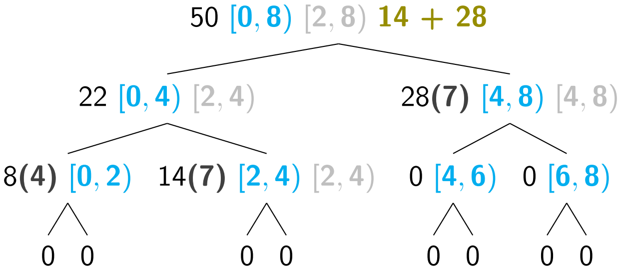


- Note how all ancestors of the node responsible for the range $[2, 4)$ have lazy counter equal to 0.

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



- 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.

```
using namespace std;

const int N = 100100;
long long lazyadd[1<<18], sum[1<<18];

// Procedure for recalculating a node's sum from its lazy and children.
void recalculate(int id, long long l, long long r) {
  sum[id] = lazyadd[id] * (r - l);
  if (r - l != 1) {
    sum[id] += sum[id * 2];
    sum[id] += sum[id * 2 + 1];
  }
}

void update_lazy(int id, long long v, long long l, long long r) {
  lazyadd[id] += v;
  recalculate(id, l, r);
}

// Preorder procedure for propagation. Do NOT call it on leaves.
void propagate(int id, long long l, long long r) {
  long long mid = (l + r) / 2;
  update_lazy(id * 2, lazyadd[id], l, mid);
  update_lazy(id * 2 + 1, lazyadd[id], mid, r);
  lazyadd[id] = 0;
}
```

```
int n;

// The root node is responsible for [0, n). Update range [uL, uR)
void update(int uL, int uR, int v, int i = 1, int cL = 0, int cR = n) {
  if (uL == cL && uR == cR) {
    update_lazy(i, v, cL, cR);
    return;
  }
  propagate(i, cL, cR);
  int mid = (cL + cR) / 2;
  if (uL < mid) update(uL, min(uR, mid), v, i * 2, cL, mid);
  if (uR > mid) update(max(uL, mid), uR, v, i * 2 + 1, mid, cR);
  recalculate(i, cL, cR);
}

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

- **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, we 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.

Example: Setting Ranges

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

Sample Input:

```
5 7
U 1 3 5
U 2 4 1
Q 1 3
Q 2 3
U 3 4 3
Q 2 4
Q 1 5
```

Sample Output:

```
5
1
3
5
```

- 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.

Example: Setting Ranges

- **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.

- 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 *i*'s children.

Example: Setting Ranges


```
using namespace std;

const int N = 100100;
const int UNSET = -1;
// Since A is 0 initially, the default values are correct.
int lazyset[1<<18]; // UNSET if no lazy is set
int maxrt[1<<18];

// Recalculates a node's values assuming its children are correct.
// do NOT call these on leaves.
void recalculate(int i) {
  if (lazyset[i] != UNSET) // should never happen
    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] = lazyset[i];
  lazyset[i] = UNSET;
}
```

Example: Setting Ranges


```
int n;

void update(int uL, int uR, int v, int i = 1, int cL = 0, int cR = n) {
  if (uL == cL && uR == cR) {
    lazyset[i] = maxrt[i] = v;
    return;
  }
  propagate(i);
  int mid = (cL + cR) / 2;
  if (uL < mid) update(uL, min(uR, mid), v, i*2, cL, mid);
  if (uR > mid) update(max(uL, mid), uR, v, i*2+1, mid, cR);
  recalculate(i);
}

int query(int qL, int qR, int i = 1, int cL = 0, int cR = n) {
  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;
}
```

Table of Contents

Range Tree of Data Structures

- So far we've just used range trees to support operations 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 `U l r d` saying that there will be rain in towns $[l, r)$ on day d. Interspersed among these updates, there will be queries of the form `Q a d`, asking if there is a predicted shower in town a on day d.
- **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 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
```

Problem: Should I bring an Umbrella?

- 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?


```
#include <set>
using namespace std;

const int N = 100100;
set<int> rt[1<<18];
int n;

// 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 cL = 0, int cR = n) {
  if (uL == cL && uR == cR) {
    rt[i].insert(v);
    return;
  }
  int mid = (cL + cR) / 2;
  if (uL < mid) update(uL, min(uR, mid), v, i * 2, cL, mid);
  if (uR > mid) update(max(uL, mid), uR, v, i * 2 + 1, mid, cR);
}

// Does it rain in index qP on day qD?
bool query(int qP, int qD, int i = 1, int cL = 0, int cR = n) {
  if (rt[i].find(qD) != rt[i].end())
    return true;
  if (cR - cL == 1)
    return false;
  int mid = (cL + cR) / 2;
  if (qP < mid) return query(qP, qD, i * 2, cL, mid);
  else return query(qP, qD, i * 2 + 1, mid, cR);
}
```

- **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.

Problem: Should I bring an Umbrella?

- We can actually do much more.
- 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.
- If the bounds were different (fewer cities) we could even support updates affecting ranges of days, by storing a range tree of days in each node of the original range tree.
- **Moral:** If you need to store different kinds of data while supporting range operations, consider a range tree of a suitable data structure.

Problem: Mapping Neptune

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

Problem: Mapping Neptune

- **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 `x0 y0 x1 y1`, 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.

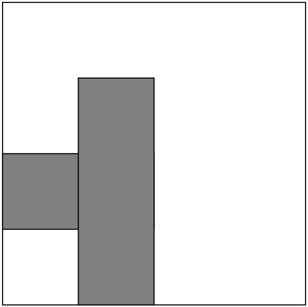
Sample Input:

```
4 2
0 1 2 2
1 0 2 3
```

Sample Output:

```
4
```

Explanation:



Problem: Mapping Neptune

- 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)

- 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


```
#include <iostream>
#include <vector>
using namespace std;

const int N = 100100;
// Range tree
int lazycount[1<<18], freq[1<<18];
int n, m;

void recompute(int i, int left, int right) {
  if (lazycount[i] > 0) freq[i] = right-left; // range directly covered
  else if (right-left == 1) freq[i] = 0; // leaf
  else freq[i] = freq[i*2] + freq[i*2+1]; // sum of children
}

// Update count of [uL, uR) by v
void update(int uL, int uR, int v, int i = 1, int cL = 0, int cR = n) {
  if (uL == cL && uR == cR) {
    lazycount[i] += v;
    recompute(i, cL, cR);
    return;
  }
  int mid = (cL + cR) / 2;
  if (uL < mid) update(uL, min(uR, mid), v, i*2, cL, mid);
  if (uR > mid) update(max(uL, mid), uR, v, i*2+1, mid, cR);
  recompute(i, cL, cR);
}

int query_total() {
  return freq[1];
}
```

Problem: Mapping Neptune


```
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 yi, we will count covered squares in [yi, yi+1]
vector<Event> events[N];

int main() {
  cin >> n >> m;
  for (int i = 0; i < m; i++) {
    int x0, y0, x1, y1;
    cin >> 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++) {
    for (const auto &e: events[i])
      update(e.l, e.r, e.v);
    ans += query_total();
  }

  cout << ans << '\n';
}
```

- 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.

- Each node in our subtree stores the answer for queries that are exactly the node's range of responsibility $[l, r)$.

- As in divide and conquer, answers contained entirely within the left half $[l, m)$ or right half $[m, r)$ of the range are handled by the left and right child.

- 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.
- A subarray crossing the midpoint will be composed of:
  - a suffix of the left half, and
  - a prefix of the right half.
- But remember, any metadata we add must itself be updated in our range tree.
  - Generally this is easier because the metadata is more specific.
  - May need to keep adding more metadata until this stabilises.

- 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 q 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, q. The following q lines each describe an operation.
- **Constraints:** $1 \leq n, q \leq 100,000$.

Sample Input:

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

Sample Output:

```
0
3
4
```

Example: Maximum Sum Subrange

- 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.

Example: Maximum Sum Subrange

- 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 `sum[i]` is the sum of *i*'s entire range.

- So we now need to maintain `sum[i]`.

- But this is easy, you've seen this many times.

- Phew! We're done now. Only needed to go 3 levels deep!

```
using namespace std;

const int MAXN = 100100;

struct state {
  long long bestStart, bestEnd, sum, bestSubarray;
};

// Default value of state is all 0. This is correct for us.
state rt[1<<18];

state mergeStates(const 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);
  /* in C++11, can instead do ret.bestSubarray = max({left.bestSubarray,
      right.bestSubarray, left.bestEnd + right.bestStart}) */
  return ret;
}
```

Example: Maximum Sum Subrange


```
int n;

void update(int p, int v, int i=1, int cL = 0, int cR = n) {
  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]);
}

state query(int qL, int qR, int i = 1, int cL = 0, int cR = n) {
  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 technique 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.

Table of Contents

Searching 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.

Searching a Range Tree

- 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: First Big Value

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

Sample Input:

```
4 7
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
```

Sample Output:

```
1
-1
0
0
```

- 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.

Problem: First Big Value


```
using namespace std;

const int N = 100100;
int maxrt[1<<18];
int n;

// Standard max range tree.
void update(int p, int v, int i = 1, int cL = 0, int cR = n) {
  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]);
}
```

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

```
bool query_rec(int qL, int qR, int v, int &foundPlc, int i = 1, int cL = 0, int
     cR = n) {
  // 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 (maxrt[i] <= v) return false;
  if (cR - cL == 1) {
    foundPlc = cL;
    return true;
  }
  int mid = (cL + cR) / 2;
  if (query_rec(qL, qR, v, foundPlc, i*2, cL, mid)) return true;
  if (query_rec(qL, qR, v, foundPlc, 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;
}
```

Problem: First Big Value

- **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 $[l_i, r_i)$ 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
```

Problem: Rising Water

- **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 > height[i] - total_water[i]`.

- So to know if there is a new town underwater, we just need to know if $\min_i(\mathit{height}[i] - \mathit{total\_water}[i]) < 0$.
- 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.

```
#include <iostream>
using namespace std;

const int N = 500500;
const int INF = 1000*1000*1000+7; // large height to never go underwater
int minrt[1<<20]; // standard range update min range tree
int n, m;

void update(int uL, int uR, int v); // standard function for a[uL, uR) += v
int query(int qL, int qR, int v); // returns index of any value < 0, or -1 if
      none exist

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 < m; i++) {
        int a, b, w, cInd;
        cin >> a >> b >> w;
        update(a, b, -w);
        while ((cInd = query(0, n, 0)) != -1) {
            ans++;
            update(cInd, cInd+1, INF); // "delete" cInd
        }
        cout << ans << '\n';
    }
}
```