<https://leetcode.com/problems/range-module/description/>

A Range Module is a module that tracks ranges of numbers. Design a data structure to track the ranges represented as **half-open intervals** and query about them.

A **half-open interval** [left, right) denotes all the real numbers x where left <= x < right.

Implement the RangeModule class:

RangeModule() Initializes the object of the data structure.

void addRange(int left, int right) Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not already tracked.

boolean queryRange(int left, int right) Returns true if every real number in the interval [left, right) is currently being tracked, and false otherwise.

void removeRange(int left, int right) Stops tracking every real number currently being tracked in the half-open interval [left, right).

**Example 1:**

**Input**["RangeModule", "addRange", "removeRange", "queryRange", "queryRange", "queryRange"][[], [10, 20], [14, 16], [10, 14], [13, 15], [16, 17]]

**Output**[null, null, null, true, false, true]

**Explanation**

RangeModule rangeModule = new RangeModule();

rangeModule.addRange(10, 20);

rangeModule.removeRange(14, 16);

rangeModule.queryRange(10, 14); // return True,(Every number in [10, 14) is being tracked)

rangeModule.queryRange(13, 15); // return False,(Numbers like 14, 14.03, 14.17 in [13, 15) are not being tracked)

rangeModule.queryRange(16, 17); // return True, (The number 16 in [16, 17) is still being tracked, despite the remove operation)

**Constraints:**

1 <= left < right <= 10^9

At most 10^4 calls will be made to addRange, queryRange, and removeRange.

**Attempt 1: 2023-12-24**

**Solution 1: Segment Tree (720min)**

**Wrong Solution**

**Some boundary condition is a problem when dealing with half-open intervals**

import java.util.\*;

public class Solution {

    class SegmentTree {

        TreeNode root;

        public SegmentTree(int start, int end) {

            root = new TreeNode(start, end);

        }

        public void update(int start, int end, boolean tracked) {

            TreeNode node = new TreeNode(start, end);

            update(root, node, tracked);

        }

        private void update(TreeNode root, TreeNode node, boolean tracked) {

            if(root == null) {

                return;

            }

            // If current node's range lies completely in update query range

            if(root.inside(node)) {

                root.tracked = tracked;

            }

            // If current node's range overlaps with update range, follow the

            // same approach as above simple update

            if(root.intersect(node)) {

                // Recur for left and right children

                int mid = root.start + (root.end - root.start) / 2;

                if(root.left == null) {

                    root.left = new TreeNode(root.start, mid);

                }

                update(root.left, node, tracked);

                if(root.right == null) {

                    root.right = new TreeNode(mid, root.end);

                }

                update(root.right, node, tracked);

                // Update current node using results of left and right calls as

                // post-order traversal

                root.tracked = root.left.tracked && root.right.tracked;

            }

        }

        public boolean query(int start, int end) {

            TreeNode node = new TreeNode(start, end);

            return query(root, node);

        }

        private boolean query(TreeNode root, TreeNode node) {

            if(root.start >= node.start && root.end < node.end) {

                return root.tracked;

            }

            if(root.start >= node.end || root.end <= node.start) {

                return true;

            }

            boolean leftQuery = (root.left == null ? root.tracked : query(root.left, node));

            boolean rightQuery = (root.right == null ? root.tracked : query(root.right, node));

            return leftQuery && rightQuery;

        }

    }

    class TreeNode {

        int start;

        int end;

        TreeNode left;

        TreeNode right;

        // Java class boolean field default is 'false',

        // no need explicitly set up to 'false'

        boolean tracked;

        public TreeNode(int start, int end) {

            this.start = start;

            this.end = end;

        }

        // Check if the current node's range fully inside a given node's range

        public boolean inside(TreeNode node) {

            if(this.start >= node.start && this.end <= node.end) {

                return true;

            }

            return false;

        }

        // Check if the current node's range intersect a given node's range

        public boolean intersect(TreeNode node) {

            if(inside(node) || this.start >= node.end || this.end <= node.start) {

                return false;

            }

            return true;

        }

    }

    SegmentTree segmentTree;

    public Solution() {

        // 1 <= left < right <= 10^9

        //segmentTree = new SegmentTree(1, (int)1e9);

        segmentTree = new SegmentTree(1, 100);

    }

    // Adds the half-open interval [left, right), tracking every

    // real number in that interval. Adding an interval that partially

    // overlaps with currently tracked numbers should add any numbers

    // in the interval [left, right) that are not already tracked.

    public void addRange(int left, int right) {

        segmentTree.update(left, right, true);

    }

    // A half-open interval [left, right) denotes all the real numbers

    // x where left <= x < right.

    // Returns true if every real number in the interval [left, right)

    // is currently being tracked, and false otherwise.

    public boolean queryRange(int left, int right) {

        return segmentTree.query(left, right);

    }

    // Stops tracking every real number currently being tracked in

    // the half-open interval [left, right).

    public void removeRange(int left, int right) {

        segmentTree.update(left, right, false);

    }

    public static void main(String[] args) {

        Solution so = new Solution();

        so.addRange(10, 20);

        so.removeRange(14, 16);

        boolean result1 = so.queryRange(10, 14); // return True,(Every number in [10, 14) is being tracked)

        boolean result2 = so.queryRange(13, 15); // return False,(Numbers like 14, 14.03, 14.17 in [13, 15) are not being tracked)

        boolean result3 = so.queryRange(16, 17); // return True, (The number 16 in [16, 17) is still being tracked, despite the remove operation)

        System.out.println("done");

    }

}

**Step by Step**

e.g

we create root as [1,100) just for simulation, instead of [1,(int)1e9)

addRange(10,20)

                                            root

                                           [1,100)

                                            false

                                  /                          \

                                left                         right

                               [1,50)                       [50,100)

                                false                        false

                        /                 \                 /     \

                      left               right            left    right

                     [1,25)              [25,50)         [50,75) [75,100)

                      false               false           false   false

                 /            \

             left              right

            [1,13)            [13,25)

             false             false

            /     \         /         \

          left   right    left        right

         [1,7)  [7,13)   [13,19)     [19,25)

         false   false    TRUE        false

                /    \                /      \

             left    right         left      right

            [7,10)  [10,13)       [19,22)   [22,25)

             false   TRUE          false     false

                                  /      \

                                left    right

                               [19,20)  [20,22)

                                TRUE     false

removeRange(14,16)

                                                             root

                                                            [1,100)

                                                             false

                                             /                                     \

                                          left                                     right

                                         [1,50)                                   [50,100)

                                          false                                    false

                               /                                \                 /     \

                             left                              right            left    right

                            [1,25)                            [25,50)         [50,75) [75,100)

                             false                             false           false   false

                 /                           \

             left                            right

            [1,13)                          [13,25)

             false                           false

            /     \                     /                     \

          left   right                left                   right

         [1,7)  [7,13)              [13,19)                 [19,25)

         false   false                T->F                   false

                /    \              /      \                /      \

             left    right       left     right          left      right

            [7,10)  [10,13)     [13,16)  [16,19)        [19,22)   [22,25)

             false   TRUE        new F    new F          false     false

                                /    \                  /      \

                             left   right             left    right

                            [13,14) [14,16)          [19,20)  [20,22)

                             new F   new F            TRUE    false

                              ? (here is the question)

The problem after removeRange(14,16) we should tag [10,14) which is actually [10,13.999...999) not include 14

with tracked, but current code still consider 14 needs to be tracked ?

Then for half open ')' should we use 'right - 1' to when add / remove ?

**Correct Solution**

class TreeNode {

    int lo;

    int hi;

    boolean tracked;

    TreeNode left;

    TreeNode right;

    public TreeNode(int lo, int hi, boolean tracked) {

        this.lo = lo;

        this.hi = hi;

        this.tracked = tracked;

    }

}

class SegmentTree {

    TreeNode root;

    public SegmentTree() {

        root = new TreeNode(1, (int)1e9, false);

    }

    public void addRange(int start, int end) {

        update(root, start, end, true);

    }

    public void removeRange(int start, int end) {

        update(root, start, end, false);

    }

    public boolean queryRange(int start, int end) {

        return query(root, start, end);

    }

    public void update(TreeNode root, int start, int end, boolean tracked) {

        // If current node's range lies completely in update query range

        if(root.lo >= start && root.hi <= end) {

            root.tracked = tracked;

            // Below two statements are required, test out by:

            // Input: ["RangeModule","addRange","removeRange","removeRange","addRange",

            // "removeRange","addRange","queryRange","queryRange","queryRange"]

            // [[],[6,8],[7,8],[8,9],[8,9],[1,3],[1,8],[2,4],[2,9],[4,6]]

            // Output = [null,null,null,null,null,null,null,false,false,false]

            // Expected = [null,null,null,null,null,null,null,true,true,true]

            root.left = null;

            root.right = null;

            return;

        }

        int mid = root.lo + (root.hi - root.lo) / 2;

        // Not like Segment Tree array solution, the TreeNode not naturally

        // exist, we have to create it when no child TreeNode, then we can

        // apply left and right branch recursive build based on child TreeNode

        if(root.left == null) {

            root.left = new TreeNode(root.lo, mid, root.tracked);

        }

        if(root.right == null) {

            root.right = new TreeNode(mid + 1, root.hi, root.tracked);

        }

        if(end <= mid) {

            update(root.left, start, end, tracked);

        } else if(start > mid) {

            update(root.right, start, end, tracked);

        } else {

            update(root.left, start, mid, tracked);

            update(root.right, mid + 1, end, tracked);

        }

        root.tracked = root.left.tracked && root.right.tracked;

    }

    public boolean query(TreeNode root, int start, int end) {

        if(root.left == null || root.right == null) {

            return root.tracked;

        }

        if(root.lo >= start && root.hi <= end) {

            return root.tracked;

        }

        int mid = root.lo + (root.hi - root.lo) / 2;

        if(end <= mid) {

            return query(root.left, start, end);

        } else if(start > mid) {

            return query(root.right, start, end);

        } else {

            return query(root.left, start, mid) && query(root.right, mid + 1, end);

        }

    }

}

class RangeModule {

    SegmentTree segmentTree;

    public RangeModule() {

        segmentTree = new SegmentTree();

    }

    public void addRange(int left, int right) {

        segmentTree.addRange(left, right - 1);

    }

    public boolean queryRange(int left, int right) {

        return segmentTree.queryRange(left, right - 1);

    }

    public void removeRange(int left, int right) {

        segmentTree.removeRange(left, right - 1);

    }

}

/\*\*

 \* Your RangeModule object will be instantiated and called as such:

 \* RangeModule obj = new RangeModule();

 \* obj.addRange(left,right);

 \* boolean param\_2 = obj.queryRange(left,right);

 \* obj.removeRange(left,right);

 \*/

Time Complexity: O(N\*logN)

Space Complexity: O(M)

The space complexity of the entire data structure is O(m) where m is the number of distinct

intervals that have been added or removed. This is because new nodes in the tree are only

created when a range is modified, which means the tree is sparse and doesn't necessarily

contain a node for every possible value within the range.

**Let's see the test code why in update() method the "root.left == null" and "root.right == null" are required.**

        // If current node's range lies completely in update query range

        if(root.lo >= start && root.hi <= end) {

            root.tracked = tracked;

            // Below two statements are required, test out by:

            // Input: ["RangeModule","addRange","removeRange","removeRange","addRange",

            // "removeRange","addRange","queryRange","queryRange","queryRange"]

            // [[],[6,8],[7,8],[8,9],[8,9],[1,3],[1,8],[2,4],[2,9],[4,6]]

            // Output = [null,null,null,null,null,null,null,false,false,false]

            // Expected = [null,null,null,null,null,null,null,true,true,true]

            root.left = null;

            root.right = null;

            return;

        }

**Test out by**

import java.util.\*;

public class Solution {

class TreeNode {

int lo;

int hi;

boolean tracked;

TreeNode left;

TreeNode right;

public TreeNode(int lo, int hi, boolean tracked) {

this.lo = lo;

this.hi = hi;

this.tracked = tracked;

}

}

class SegmentTree {

TreeNode root;

public SegmentTree() {

root = new TreeNode(1, (int)20, false);

}

public void addRange(int start, int end) {

update(root, start, end, true);

}

public void removeRange(int start, int end) {

update(root, start, end, false);

}

public boolean queryRange(int start, int end) {

return query(root, start, end);

}

public void update(TreeNode root, int start, int end, boolean tracked) {

// If current node's range lies completely in update query range

if(root.lo >= start && root.hi <= end) {

root.tracked = tracked;

// Below two statements are required, test out by:

// Input: ["RangeModule","addRange","removeRange","removeRange","addRange",

// "removeRange","addRange","queryRange","queryRange","queryRange"]

// [[],[6,8],[7,8],[8,9],[8,9],[1,3],[1,8],[2,4],[2,9],[4,6]]

// Output = [null,null,null,null,null,null,null,false,false,false]

// Expected = [null,null,null,null,null,null,null,true,true,true]

root.left = null;

root.right = null;

return;

}

int mid = root.lo + (root.hi - root.lo) / 2;

// Not like Segment Tree array solution, the TreeNode not naturally

// exist, we have to create it when no child TreeNode, then we can

// apply left and right branch recursive build based on child TreeNode

if(root.left == null) {

root.left = new TreeNode(root.lo, mid, root.tracked);

}

if(root.right == null) {

root.right = new TreeNode(mid + 1, root.hi, root.tracked);

}

if(end <= mid) {

update(root.left, start, end, tracked);

} else if(start > mid) {

update(root.right, start, end, tracked);

} else {

update(root.left, start, mid, tracked);

update(root.right, mid + 1, end, tracked);

}

root.tracked = root.left.tracked && root.right.tracked;

}

public boolean query(TreeNode root, int start, int end) {

if(root.left == null || root.right == null) {

return root.tracked;

}

if(root.lo >= start && root.hi <= end) {

return root.tracked;

}

int mid = root.lo + (root.hi - root.lo) / 2;

if(end <= mid) {

return query(root.left, start, end);

} else if(start > mid) {

return query(root.right, start, end);

} else {

return query(root.left, start, mid) && query(root.right, mid + 1, end);

}

}

}

SegmentTree segmentTree;

public Solution() {

segmentTree = new SegmentTree();

}

public void addRange(int left, int right) {

segmentTree.addRange(left, right - 1);

}

public boolean queryRange(int left, int right) {

return segmentTree.queryRange(left, right - 1);

}

public void removeRange(int left, int right) {

segmentTree.removeRange(left, right - 1);

}

public static void main(String[] args) {

// Input: ["RangeModule","addRange","removeRange","removeRange","addRange",

// "removeRange","addRange","queryRange","queryRange","queryRange"]

// a r r a r a q q q

// [[],[6,8],[7,8],[8,9],[8,9],[1,3],[1,8],[2,4],[2,9],[4,6]]

// Output = [null,null,null,null,null,null,null,false,false,false]

// Expected = [null,null,null,null,null,null,null,true,true,true]

Solution so = new Solution();

so.addRange(6, 8);

so.removeRange(7, 8);

so.removeRange(8, 9);

so.addRange(8, 9);

so.removeRange(1, 3);

so.addRange(1, 8);

boolean result1 = so.queryRange(2, 4); // return True,(Every number in [10, 14) is being tracked)

boolean result2 = so.queryRange(2, 9); // return False,(Numbers like 14, 14.03, 14.17 in [13, 15) are not being tracked)

boolean result3 = so.queryRange(4, 6); // return True, (The number 16 in [16, 17) is still being tracked, despite the remove operation)

System.out.println("done");

}

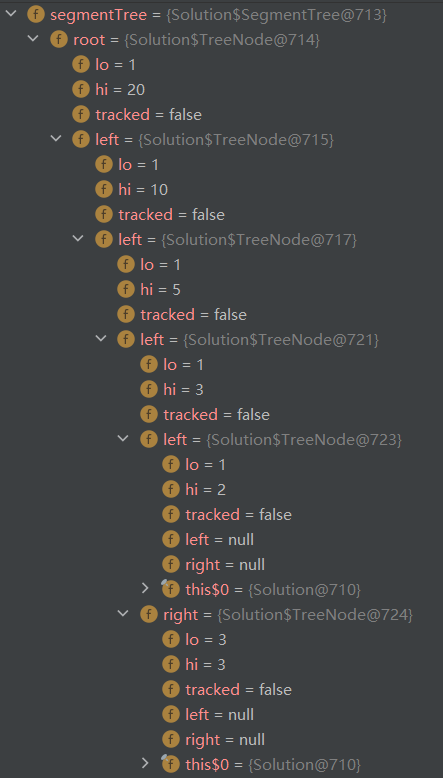
}

We observe the deviation between correct solution which includes "root.left == null" and "root.right == null" in update() method happen after two statements:

so.removeRange(1, 3);

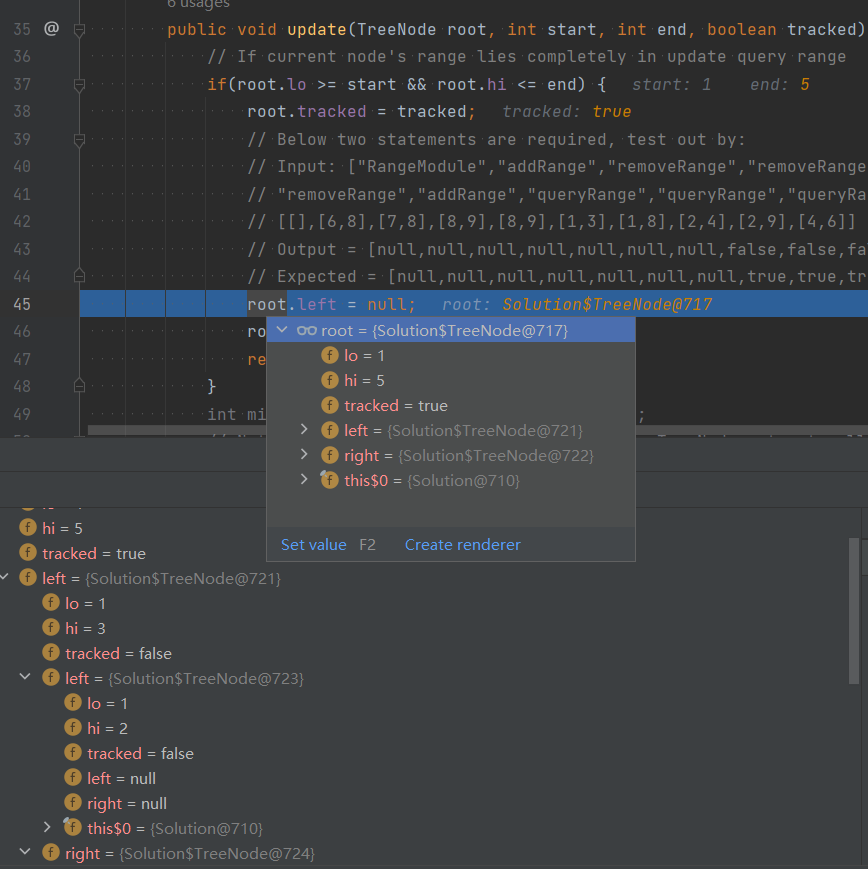
so.addRange(1, 8);

In correct solution after so.removeRange(1, 3);

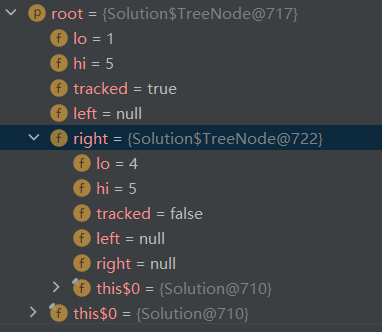


**Then after we call so.addRange(1, 8);, we observe root(1,5,false)'s first change to (1,5,true), then its left child (1,3,false) and right child (4,5,false) both get removed, this will make sure all node status keep alignment when update to 'true', like the example here if we mark range (1,5) tracked, but its subrange (1,3) and (4,5) still mark as not tracked because of previous so.removeRange(1,3), that is the conflict between, so when we update one node's tracked status to 'true', to make it sychronous, we have to remove its left and right children nodes to stop the further queries (so.queryRange(...)) hit its children nodes whose status keep as 'false' (Note: this might be more efficient than update all children nodes's tracked status till leaf to 'true')**

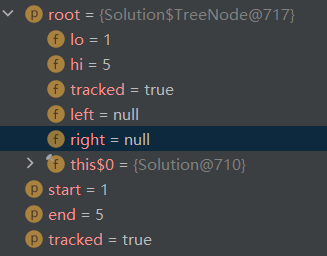
**(1) root(1,5,false)'s first change to (1,5,true)**



**(2) left child (1,3,false) get removed**



**(3) right child (4,5,false) get removed**



**After above steps, when we call so.queryRange(2, 4), since it drops inside the root(1, 5, true), and no further hit on removed subroot(1,3,false) or subroot(4,5,false), we will directly return tracked status as 'true', but if no statements like root.left = null and root.right = null, the subroot(1,3,false) and subroot(4,5,false) will still exist and the so.queryRange(2, 4)will further hit these subroots and return wrong tracked status as 'false'**

**Refer to**

<https://algo.monster/liteproblems/715>

**Problem Description**

A Range Module is asked to manage and track ranges of numbers within its data structure, specifically defined as half-open intervals [left, right). A half-open interval includes the left boundary and excludes the right boundary, which could represent all real numbers x where left <= x < right. The Range Module will support adding new ranges, querying if a given range is fully tracked, and removing existing ranges.

We have to implement a RangeModule class with the following functions:

RangeModule() - a constructor method to initialize the range tracking data structure.

addRange(int left, int right) - this method should add a half-open interval [left, right), tracking all the real numbers within that interval. If the new interval overlaps with any currently tracked numbers, only those not already tracked should be added.

queryRange(int left, int right) - this method should return true if every real number in the interval [left, right) is currently tracked. Otherwise, it returns false.

removeRange(int left, int right) - this method should stop tracking any real number in the half-open interval [left, right) that is currently being tracked.

The best way to manage these ranges effectively, given that the intervals could be large, is to use an interval tree or segment tree data structure.

**Intuition**

To solve this problem efficiently, we need a data structure that can handle dynamic range updates and queries. The segment tree is an ideal choice here due to its ability to maintain information about intervals and support operations and queries in logarithmic time complexity.

A segment tree is a tree-like data structure that stores aggregates for a range of array indices. In this context, we're interested in tracking if a range is fully added, not added, or partially added. To accommodate these conditions and to facilitate lazy propagation of updates (deferred tree updates until needed to save time), we extend the segment tree with flags that indicate the addition or removal of ranges.

Here's the breakdown for the intuition behind the proposed solution:

**Segment Tree Initialization:** Construct a Segment Tree with a sufficient range to cover the possible inputs. We start with a root node representing the entire range.

**Add and Remove Ranges:** When adding or removing a range, the tree nodes corresponding to that interval are updated. If a node's interval is entirely covered by the range, its flag is updated, and so is the v property indicating whether the range is fully added or removed.

**Lazy Propagation:** To conserve operations, we propagate changes only when necessary (when we need to go deeper into an interval that was previously updated but not propagated). We update child nodes only when a query or further update for a sub-interval within the affected range occurs.

**Querying Ranges:** To check whether an interval is fully tracked, we inspect the tree nodes associated with that interval. If all parts demonstrate that they are fully tracked, we return true; otherwise, we return false.

The Segment Tree used in the code is slightly specialized for this problem. Instead of maintaining a count or sum over an interval, each node in the Segment Tree stores whether the interval is fully included in the tracking (node.v) and a propagation flag (node.add) that indicates whether an add or remove operation should be propagated to the node's children.

This approach efficiently supports the three operations required by the Range Module: adding, querying, and removing ranges. The time complexity for each operation is O(log n), making it appropriate for problems with large ranges of numbers and many operations.

**Solution Approach**

The solution uses a Segment Tree to manage the intervals, which inherently uses a divide and conquer strategy. A Segment Tree is a binary tree data structure where each node represents an interval. For this problem, we use the Segment Tree to track whether intervals are fully included (node.v) or not, and we make sure to propagate any changes if necessary through the use of a lazy propagation technique (node.add).

Here's a step-by-step breakdown of the key elements of the Segment Tree as implemented in this solution:

**Node Structure:** Each Node of the Segment Tree has two child nodes (left and right), the add property, and a boolean value v. The add property indicates the pending operation that should be applied to this node's range (1 for adding, -1 for removing), and the v property shows whether the current interval is completely added.

**Segment Tree Initialization:** The SegmentTree class has a root node that initially represents the entire range [1, 1e9).

**The modify Function:** This function recursively updates nodes in the Segment Tree to reflect added or removed intervals. It receives the interval (left, right) and a value v indicating whether to add (1) or remove (-1). The function:

Updates the current node if its interval is completely within the range.

Calls pushdown to propagate any pending updates to the children.

Recursively calls itself to update the children nodes if they overlap with the range.

Calls pushup to update the parent node based on the update results from the children.

**Lazy Propagation (pushdown and pushup Functions):** The lazy propagation is handled by two functions:

pushdown: It takes a node and propagates the pending add operation, if any, to the children nodes, ensuring that further recursive calls will operate on updated intervals.

pushup: It updates the node.v based on the values of the children, setting it to True only if both children intervals are fully tracked.

**The query Function:** Checks if the interval is completely tracked within the range. It recursively checks nodes overlapping with the given interval and returns True only if all parts of the interval are fully tracked.

**RangeModule Class:** This class utilizes the SegmentTree and exposes the addRange, queryRange, and removeRange methods.

addRange(left, right): It calls the modify function of the tree to add the interval [left, right - 1) since the interval is half-open.

queryRange(left, right): It calls the query function of the tree to check if the interval [left, right - 1) is fully tracked.

removeRange(left, right): Similar to addRange, it calls the modify function but with the intention to remove the interval by using -1 as the value.

This approach is efficient and scalable due to the nature of Segment Trees in handling a large number of interval-based operations with time complexity in O(log n), which is due to the tree's balanced structure.

**Example Walkthrough**

Let's illustrate how the RangeModule works with a simple example. Consider a Range Module that initially tracks no numbers. We'll perform a sequence of operations and see how the internal data structure (a Segment Tree in this case) deals with these operations.

**Example Scenario:**

addRange(5, 8): Adds the interval [5, 8) to the tracking. Internally, the Segment Tree will update the nodes corresponding to this range, setting the node.v to True for this interval and using lazy propagation if necessary.

queryRange(3, 7): Queries whether the interval [3, 7) is entirely tracked. The tree checks the nodes that cover this range. Since the interval [5, 7) within the query is tracked but [3, 5) is not, the query function returns False.

addRange(7, 10): Adds the interval [7, 10) to the tracking. The tree updates nodes from 7 to 10. After this addition, the interval [5, 10) is now fully tracked.

queryRange(5, 10): Now, when querying this interval, the query function will check the nodes covering [5, 10). Since both [5, 8) and [7, 10) are tracked (with overlap), the query returns True, confirming the entire range is tracked.

removeRange(5, 6): Stops tracking the interval [5, 6). The tree updates the relevant nodes, setting node.v to False for this range and using lazy propagation to defer updates to its child nodes until needed.

queryRange(5, 10): After the removal, a query for [5, 10) would once again return False because not all numbers in this range are tracked. Specifically, the interval [5, 6) is not tracked anymore.

The Segment Tree efficiently handles these updates and queries, allowing for scalable range management without having to traverse the entire range every time we perform an operation. This is why the structure is particularly well-suited for the Range Module functionality, which may involve a large number of operations on potentially large ranges. Each operation - adding, querying, and removing - works with a time complexity of O(log n), where n is the number of nodes in the tree, which corresponds to the interval lengths being managed.

// Node class to represent each segment in the Segment Tree

class Node {

    Node left;

    Node right;

    int valueToAdd;

    boolean isCovered;

}

// SegmentTree class encapsulating the tree operations

class SegmentTree {

    private Node root = new Node();

    // Constructor is not necessary, Java provides a default one

    // Update the segment tree, marking ranges

    public void modify(int left, int right, int value) {

        modify(left, right, value, 1, (int) 1e9, root);

    }

    // Helper function to modify nodes in the given range [left, right] recursively

    private void modify(int left, int right, int value, int start, int end, Node node) {

        if (start >= left && end <= right) {

            node.isCovered = value == 1;

            node.valueToAdd = value;

            return;

        }

        pushdown(node);

        int mid = (start + end) >> 1; // Equivalent to dividing by 2

        if (left <= mid) {

            modify(left, right, value, start, mid, node.left);

        }

        if (right > mid) {

            modify(left, right, value, mid + 1, end, node.right);

        }

        pushup(node);

    }

    // Query the segment tree to check coverage of range [left, right]

    public boolean query(int left, int right) {

        return query(left, right, 1, (int) 1e9, root);

    }

    // Helper function to query the tree recursively

    private boolean query(int left, int right, int start, int end, Node node) {

        if (node == null) {

            return false;

        }

        if (start >= left && end <= right) {

            return node.isCovered;

        }

        pushdown(node);

        int mid = (start + end) >> 1;

        boolean result = true;

        if (left <= mid) {

            result = result && query(left, right, start, mid, node.left);

        }

        if (right > mid) {

            result = result && query(left, right, mid + 1, end, node.right);

        }

        return result;

    }

    // Function to update node coverage based on child nodes

    private void pushup(Node node) {

        if (node.left != null && node.right != null) {

            node.isCovered = node.left.isCovered && node.right.isCovered;

        }

    }

    // Function to propagate changes to child nodes

    private void pushdown(Node node) {

        if (node.left == null) node.left = new Node();

        if (node.right == null) node.right = new Node();

        if (node.valueToAdd != 0) {

            // Propagate valueToAdd down to children and reset it for the current node

            node.left.valueToAdd = node.valueToAdd;

            node.right.valueToAdd = node.valueToAdd;

            node.left.isCovered = node.valueToAdd == 1;

            node.right.isCovered = node.valueToAdd == 1;

            node.valueToAdd = 0;

        }

    }

}

// RangeModule class to handle range adding, querying, and removing

class RangeModule {

    private SegmentTree tree = new SegmentTree();

    // Adds a range to the module

    public void addRange(int left, int right) {

        tree.modify(left, right - 1, 1);

    }

    // Queries if the range is fully covered

    public boolean queryRange(int left, int right) {

        return tree.query(left, right - 1);

    }

    // Removes a range from the module

    public void removeRange(int left, int right) {

        tree.modify(left, right - 1, -1);

    }

}

/\*\*

 \* Your RangeModule object will be instantiated and called as follows:

 \* RangeModule obj = new RangeModule();

 \* obj.addRange(left, right);

 \* boolean param\_2 = obj.queryRange(left, right);

 \* obj.removeRange(left, right);

 \*/

**Time and Space Complexity**

**Time Complexity:**

For the SegmentTree methods, let's analyze the time complexity:

modify(left, right, v):

The time complexity of this function can be considered as O(log(range)) where range is the maximum value that the segment tree handles (in this case, range is int(1e9) by default). Despite the large range, actual complexity depends on the height of the segment tree given by O(log(n)) where n is the number of intervals added or removed since the tree becomes finer only in the areas that are modified.

query(left, right):

The time complexity is O(log(range)) for the same reasons as the modify function, with actual complexity depending on the interval's distribution and the number of nodes created in those regions.

pushup(node):

This has a constant time complexity of O(1) as it is just updating the value of the current node based on its children.

pushdown(node):

Also operates in O(1) in terms of the direct operation it performs, though it can trigger the lazy creation of child nodes not previously accessed.

In the context of the RangeModule:

addRange(left, right), queryRange(left, right), removeRange(left, right) each make a call to modify(left, right, v) and query(left, right) on the SegmentTree, so they inherit the same time complexity.

**Space Complexity:**

The space complexity of the entire data structure is O(m) where m is the number of distinct intervals that have been added or removed. This is because new nodes in the tree are only created when a range is modified, which means the tree is sparse and doesn't necessarily contain a node for every possible value within the range.

Please note, the complexity analysis is given under the assumption that ranges do not tightly cover the entire possible range, meaning it is assuming sparsity typical of use-cases for segment trees, which allows us to consider complexities in terms of the number of modifications (n or m) rather than the total covered range (int(1e9)).

**Refer to**

<https://leetcode.com/problems/range-module/solutions/1299690/java-segment-tree-with-comments/>

Several details that got me stuck:

The right-open intercal [left, right). This is taken care of by using right - 1 as input of the functions.

When it reaches the exact target interval, update it and remove all inner status since everything in the target interval has been taken care of.

Always create left and right children of a node at the same time.

Make leftTracked and rightTracked true by default as there are cases that the interval only falls in either [start, mid] or [mid + 1, end].

class RangeModule {

private SegmentTree tree;

public RangeModule() {

tree = new SegmentTree(1, (int) 1e9);

}

// the right-open interval is implemented by right - 1

public void addRange(int left, int right) {

tree.modify(left, right - 1, true);

}

public boolean queryRange(int left, int right) {

return tree.query(left, right - 1);

}

public void removeRange(int left, int right) {

tree.modify(left, right - 1, false);

}

}

/\*\*

\* Your RangeModule object will be instantiated and called as such:

\* RangeModule obj = new RangeModule();

\* obj.addRange(left,right);

\* boolean param\_2 = obj.queryRange(left,right);

\* obj.removeRange(left,right);

\*/

class SegmentTreeNode{

public int start, end;

public boolean tracked;

public SegmentTreeNode left, right;

public SegmentTreeNode(int start, int end, boolean track){

this.start = start;

this.end = end;

left = right = null;

tracked = track;

}

}

class SegmentTree{

private SegmentTreeNode root;

// constructor

public SegmentTree(int start, int end){

root = new SegmentTreeNode(start, end, false);

}

// public func to add or remove an interval

public void modify(int start, int end, boolean track){

modify(root, start, end, track);

}

// private overload func

private void modify(SegmentTreeNode node, int start, int end, boolean track){

if(node.start == start && node.end == end){

node.tracked = track;

node.left = node.right = null; // update out all nodes in between

return;

}

int mid = node.start + (node.end - node.start) / 2;

// if left and right subtree does not exist

if(node.left == null){

node.left = new SegmentTreeNode(node.start, mid, node.tracked);

node.right = new SegmentTreeNode(mid + 1, node.end, node.tracked);

}

if(start <= mid){

modify(node.left, start, Math.min(end, mid), track);

}

if(end >= mid + 1){

modify(node.right, Math.max(start, mid + 1), end, track);

}

node.tracked = node.left.tracked & node.right.tracked;

}

public boolean query(int start, int end){

return query(root, start, end);

}

private boolean query(SegmentTreeNode node, int start, int end){

// not splits in between

if(node.left == null) return node.tracked;

if(node.start == start && node.end == end){

return node.tracked;

}

int mid = node.start + (node.end - node.start) / 2;

boolean leftTracked = true, rightTracked = true;

// set the default to be true for one half range cases

if(start <= mid){

leftTracked = query(node.left, start, Math.min(end, mid));

}

if(end >= mid + 1){

rightTracked = query(node.right, Math.max(mid + 1, start), end);

}

return leftTracked & rightTracked;

}

}

**下面的答案是最接近L307模版体系的答案**

**Refer to**

<https://walkccc.me/LeetCode/problems/0715/#__tabbed_2_2>

**C++ version**

struct SegmentTreeNode {

int lo;

int hi;

bool tracked = false;

SegmentTreeNode\* left;

SegmentTreeNode\* right;

SegmentTreeNode(int lo, int hi, bool tracked, SegmentTreeNode\* left = nullptr,

SegmentTreeNode\* right = nullptr)

: lo(lo), hi(hi), tracked(tracked), left(left), right(right) {}

~SegmentTreeNode() {

delete left;

delete right;

left = nullptr;

right = nullptr;

}

};

class SegmentTree {

public:

explicit SegmentTree() : root(make\_unique<SegmentTreeNode>(0, 1e9, false)) {}

void addRange(int i, int j) {

update(root.get(), i, j, true);

}

bool queryRange(int i, int j) {

return query(root.get(), i, j);

}

void removeRange(int i, int j) {

update(root.get(), i, j, false);

}

private:

std::unique\_ptr<SegmentTreeNode> root;

void update(SegmentTreeNode\* root, int i, int j, bool tracked) {

if (root->lo == i && root->hi == j) {

root->tracked = tracked;

root->left = nullptr;

root->right = nullptr;

return;

}

const int mid = root->lo + (root->hi - root->lo) / 2;

if (root->left == nullptr) {

root->left = new SegmentTreeNode(root->lo, mid, root->tracked);

root->right = new SegmentTreeNode(mid + 1, root->hi, root->tracked);

}

if (j <= mid)

update(root->left, i, j, tracked);

else if (i > mid)

update(root->right, i, j, tracked);

else {

update(root->left, i, mid, tracked);

update(root->right, mid + 1, j, tracked);

}

root->tracked = root->left->tracked && root->right->tracked;

}

bool query(SegmentTreeNode\* root, int i, int j) {

if (root->left == nullptr)

return root->tracked;

if (root->lo == i && root->hi == j)

return root->tracked;

const int mid = root->lo + (root->hi - root->lo) / 2;

if (j <= mid)

return query(root->left, i, j);

if (i > mid)

return query(root->right, i, j);

return query(root->left, i, mid) && query(root->right, mid + 1, j);

}

};

class RangeModule {

public:

void addRange(int left, int right) {

tree.addRange(left, right - 1);

}

bool queryRange(int left, int right) {

return tree.queryRange(left, right - 1);

}

void removeRange(int left, int right) {

tree.removeRange(left, right - 1);

}

private:

SegmentTree tree;

};

**Java version (convert from ChatGPT)**

Please note that in Java, we use null instead of nullptr, and the bool type is replaced with the boolean type. Additionally, the unique\_ptr is not needed in Java since Java handles memory management automatically with garbage collection.

class SegmentTreeNode {

int lo;

int hi;

boolean tracked;

SegmentTreeNode left;

SegmentTreeNode right;

SegmentTreeNode(int lo, int hi, boolean tracked, SegmentTreeNode left, SegmentTreeNode right) {

this.lo = lo;

this.hi = hi;

this.tracked = tracked;

this.left = left;

this.right = right;

}

}

class SegmentTree {

private SegmentTreeNode root;

SegmentTree() {

this.root = new SegmentTreeNode(0, (int) 1e9, false, null, null);

}

void addRange(int i, int j) {

update(root, i, j, true);

}

boolean queryRange(int i, int j) {

return query(root, i, j);

}

void removeRange(int i, int j) {

update(root, i, j, false);

}

private void update(SegmentTreeNode root, int i, int j, boolean tracked) {

if (root.lo == i && root.hi == j) {

root.tracked = tracked;

root.left = null;

root.right = null;

return;

}

int mid = root.lo + (root.hi - root.lo) / 2;

if (root.left == null) {

root.left = new SegmentTreeNode(root.lo, mid, root.tracked, null, null);

root.right = new SegmentTreeNode(mid + 1, root.hi, root.tracked, null, null);

}

if (j <= mid)

update(root.left, i, j, tracked);

else if (i > mid)

update(root.right, i, j, tracked);

else {

update(root.left, i, mid, tracked);

update(root.right, mid + 1, j, tracked);

}

root.tracked = root.left.tracked && root.right.tracked;

}

private boolean query(SegmentTreeNode root, int i, int j) {

if (root.left == null)

return root.tracked;

if (root.lo == i && root.hi == j)

return root.tracked;

int mid = root.lo + (root.hi - root.lo) / 2;

if (j <= mid)

return query(root.left, i, j);

if (i > mid)

return query(root.right, i, j);

return query(root.left, i, mid) && query(root.right, mid + 1, j);

}

}

class RangeModule {

private SegmentTree tree;

RangeModule() {

this.tree = new SegmentTree();

}

void addRange(int left, int right) {

tree.addRange(left, right - 1);

}

boolean queryRange(int left, int right) {

return tree.queryRange(left, right - 1);

}

void removeRange(int left, int right) {

tree.removeRange(left, right - 1);

}

}

**本题的重点在L307模版的基础上从Segment Tree的Array表达升级为TreeNode表达，以下是L307的Segment Tree的Array表达，而递归方法中各种条件判断模式几乎一致，可以作为参考：**

// Style 2: Recursive version of Segment Trees

class NumArray {

int[] tree;

int n;

public NumArray(int[] nums) {

n = nums.length;

// Why not n \* 2 but n \* 4 ?

// https://stackoverflow.com/questions/28470692/how-is-the-memory-of-the-array-of-segment-tree-2-2-ceillogn-1

tree = new int[4 \* n];

// Call this method as buildTree(nums, 0, 0, n-1),

// Here nums[] is input array and n is its size

buildTree(nums, 0, 0, n - 1);

}

// Pending:

// 1. Modify to root.left = ... root.right = ... Divide and Conquer way which has return type

// 2. Start with index = 1, not index = 0, then child = 2 \* i & 2 \* i + 1

// --------------------------------------------------------------------------------

// The build tree process is quite different than iterative segment tree, which

// directly assign original array into tree as leaves, indexes between [n, 2n - 1],

// in recursive solution, the leaves node not directly locate between [n, 2n - 1],

// instead follow 2 \* treeIndex + 1 or 2 \* treeIndex + 2 rule level by level

// until the leaves level reached, that's also why instead only requires 2 \* n

// length for iterative segement tree, in recursive segement tree requires 4 \* n

private void buildTree(int[] nums, int treeIndex, int lo, int hi) {

if(lo == hi) {

tree[treeIndex] = nums[lo];

return;

}

int mid = lo + (hi - lo) / 2;

buildTree(nums, 2 \* treeIndex + 1, lo, mid);

buildTree(nums, 2 \* treeIndex + 2, mid + 1, hi);

tree[treeIndex] = tree[2 \* treeIndex + 1] + tree[2 \* treeIndex + 2];

}

public void update(int index, int val) {

// Call this method as updateHelper(0, 0, n-1, i, val),

// here you want to update the value at index with value val.

updateHelper(0, 0, n - 1, index, val);

}

private void updateHelper(int treeIndex, int lo, int hi, int index, int val) {

// Leaf node, update element

if(lo == hi) {

tree[treeIndex] = val;

return;

}

// Recurse deeper for appropriate child

int mid = lo + (hi - lo) / 2;

if(index > mid) {

updateHelper(2 \* treeIndex + 2, mid + 1, hi, index, val);

} else {

updateHelper(2 \* treeIndex + 1, lo, mid, index, val);

}

// Merge updates

tree[treeIndex] = tree[2 \* treeIndex + 1] + tree[2 \* treeIndex + 2];

}

public int sumRange(int left, int right) {

// Call this method as querySegTree(0, 0, n-1, left, right),

// here [left, right] is the range/interval you are querying.

// This method relies on "null" nodes being equivalent to storing zero.

return sumRangeHelper(0, 0, n - 1, left, right);

}

// Query for arr[left ... right]

// Note: the coordinate as [left, right] is the basement, which is part of

// segment tree, the variable keep on changing and should compare with the

// [left, right] range as boundary judgement is [lo, hi]

private int sumRangeHelper(int treeIndex, int lo, int hi, int left, int right) {

// Segment completely outside range, represents a null node

// Basement: --- left --- right --- OR --- left --- right ---

// Variable: ---------------------- lo ------ hi ----------------------

if(lo > right || hi < left) {

return 0;

}

// Segment completely inside range

// Basement: --- left ------------ right ---

// Variable: ---------------------- lo ------ hi ----------------------

if(lo >= left && hi <= right) {

return tree[treeIndex];

}

// Partial overlap of current segment and queried range. Recurse deeper

int mid = lo + (hi - lo) / 2;

if(left > mid) {

return sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, left, right);

} else if(right <= mid) {

return sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, right);

} else {

int leftSum = sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, mid);

int rightSum = sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, mid + 1, right);

return leftSum + rightSum;

}

}

}