Range Module

A Range Module is a module that tracks ranges of numbers. Your task is to design and implement the following interfaces in an efficient manner.

- `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.
- `queryRange(int left, int right)` Returns true if and only if every real number in the interval `[left, right)` is currently being tracked.
- `removeRange(int left, int right)` Stops tracking every real number currently being tracked in the interval `[left, right)`.

Example 1:

```
addRange(10, 20): null
removeRange(14, 16): null
queryRange(10, 14): true (Every number in [10, 14) is being tracked)
queryRange(13, 15): false (Numbers like 14, 14.03, 14.17 in [13, 15) are not being tr
acked)
queryRange(16, 17): true (The number 16 in [16, 17) is still being tracked, despite t
he remove operation)
```

Note:

- A half open interval `[left, right)` denotes all real numbers `left .`
- `0 in all calls to addRange, queryRange, removeRange.`
- The total number of calls to `addRange` in a single test case is at most `1000`.
- The total number of calls to `queryRange` in a single test case is at most `5000`.
- The total number of calls to `removeRange` in a single test case is at most `1000`.

Solution 1

The solution using vector of intervals (pair<int, int>) is very straightforward. The runtime is O(n) for addRange and deleteRange, and O(logn) for queryRange, where n is total number of intervals.

Another option is to use map (ordered map). And the runtime is still O(logn) for queryRange, but O(klogn) for addRange and deleteRange, where k is number of overlapping ranges.

For a single operation, it is hard to tell whether vector or map is better, because k is unknown. However, the k overlapping ranges will be erased after either add or remove ranges. Let's assume m is the total operations of add or delete ranges. Then total number of possible ranges is O(m) because add or delete may increase ranges by 1. So both n and k is O(m).

In summary, the run time for query is the same. However, the total run time for add and delete using vector is O(m^2), and that using map is O(mlogm). So amortized cost for delete and add is O(m) for vector, and O(logm) for map.

Vector

```
class RangeModule {
public:
   void addRange(int left, int right) {
        int n = invals.size();
        vector<pair<int, int>> tmp;
        for (int i = 0; i <= n; i++) {
            if (i == n || invals[i].first > right) {
                tmp.push_back({left, right});
                while (i < n) tmp.push_back(invals[i++]);
            }
            else if (invals[i].second < left)
                tmp.push_back(invals[i]);
            else {
                left = min(left, invals[i].first);
                right = max(right, invals[i].second);
            }
        }
        swap(invals, tmp);
    }

    bool queryRange(int left, int right) {
        int n = invals.size(), l = 0, r = n-1;
        while (l <= r) {
            int m = l+(r-l)/2;
            if (invals[m].first >= right)
                r = m-1;
            else if (invals[m].second <= left)
                l = m+1;
            else
                return invals[m].first <= left && invals[m].second >= right;
        }
        return false;
    }

    void removeRange(int left, int right) {
        int n = invals.size();
        vector<pair<int, int>> tmp;
        for (int i = 0; i < n; i++) {
            if (invals[i].second <= left || invals[i].first >= right)
                tmp.push_back(invals[i]);
            else {
                if (invals[i].first < left)  tmp.push_back({invals[i].first, left})
;
                if (invals[i].second > right) tmp.push_back({right, invals[i].secon
d});
            }
        }
        swap(invals, tmp);
    }
private:
    vector<pair<int, int>> invals;
};
```

Using map

```cpp
class RangeModule {
public:
    void addRange(int left, int right) {
        auto l = invals.upper_bound(left), r = invals.upper_bound(right);
        if (l != invals.begin()) {
            l--;
            if (l->second < left) l++;
        }
        if (l != r) {
            left = min(left, l->first);
            right = max(right, (--r)->second);
            invals.erase(l,++r);
        }
        invals[left] = right;
    }

    bool queryRange(int left, int right) {
        auto it = invals.upper_bound(left);
        if (it == invals.begin() || (--it)->second < right) return false;
        return true;
    }

    void removeRange(int left, int right) {
        auto l = invals.upper_bound(left), r = invals.upper_bound(right);
        if (l != invals.begin()) {
            l--;
            if (l->second < left) l++;
        }
        if (l == r) return;
        int l1 = min(left, l->first), r1 = max(right, (--r)->second);
        invals.erase(l, ++r);
        if (l1 < left) invals[l1] = left;
        if (r1 > right) invals[right] = r1;
    }
private:
    map<int, int> invals;
};
```

written by zestypanda original link here

Solution 2

`self.X` is a sorted list of x-coordinates used by add/remove, where the tracking might start/stop. The corresponding `self.track` values tell whether tracking is on at the coordinate and to its right. Removing is really just adding a range of False, so I reuse addRange for it.

```
class RangeModule(object):

    def __init__(self):
        self.X = [0, 10**9]
        self.track = [False] * 2

    def addRange(self, left, right, track=True):
        def index(x):
            i = bisect.bisect_left(self.X, x)
            if self.X[i] != x:
                self.X.insert(i, x)
                self.track.insert(i, self.track[i-1])
            return i
        i = index(left)
        j = index(right)
        self.X[i:j] = [left]
        self.track[i:j] = [track]

    def queryRange(self, left, right):
        i = bisect.bisect(self.X, left) - 1
        j = bisect.bisect_left(self.X, right)
        return all(self.track[i:j])

    def removeRange(self, left, right):
        self.addRange(left, right, False)
```

written by StefanPochmann original link here

Solution 3

```java
class RangeModule {
    private class Interval {
        int start, end;
        public Interval(int start, int end) {
            this.start = start;
            this.end = end;
        }
    }
    TreeMap<Integer, Interval> intervals;
    public RangeModule() {
        intervals = new TreeMap<>();
    }

    public void addRange(int left, int right) {
        if (intervals.containsKey(left)) {
            Interval cur = intervals.get(left);
            cur.end = Math.max(cur.end, right);
            Map.Entry<Integer, Interval> high = intervals.higherEntry(cur.start);
            while (high != null) {
                if (high.getKey() > cur.end) break;
                intervals.remove(high.getKey());
                cur.end = Math.max(cur.end, high.getValue().end);
                high = intervals.higherEntry(cur.start);
            }
        } else {
            Interval cur = new Interval(left, right);
            Map.Entry<Integer, Interval> low = intervals.lowerEntry(left);
            if (low != null && low.getValue().end >= cur.start) {
                intervals.remove(low.getKey());
                cur.start = Math.min(cur.start, low.getValue().start);
                cur.end = Math.max(cur.end, low.getValue().end);
            }
            Map.Entry<Integer, Interval> high = intervals.higherEntry(cur.start);
            while (high != null) {
                if (high.getKey() > cur.end) break;
                intervals.remove(high.getKey());
                cur.end = Math.max(cur.end, high.getValue().end);
                high = intervals.higherEntry(cur.start);
            }
            intervals.put(cur.start, cur);
        }
    }

    public boolean queryRange(int left, int right) {
        if (intervals.containsKey(left)) {
            Interval cur = intervals.get(left);
            return cur.end >= right;
        } else {
            Map.Entry<Integer, Interval> low = intervals.lowerEntry(left);
            return low != null && low.getValue().end >= right;
        }
    }

    public void removeRange(int left, int right) {
        if (intervals.containsKey(left)) {
            Interval cur = intervals.get(left);
            while (cur != null) {
                if (cur.start >= right) break;
                intervals.remove(cur.start);
                if (right <= cur.end) {
                    cur.start = right;
                    intervals.put(cur.start, cur);
                    break;
                } else {
                    Map.Entry<Integer, Interval> high = intervals.higherEntry(cur.s
tart);
                    cur = high == null ? null : high.getValue();
                }
            }
        } else {
            Map.Entry<Integer, Interval> low = intervals.lowerEntry(left);
            if (low != null) {
                if (right < low.getValue().end) {
                    intervals.put(right, new Interval(right, low.getValue().end));
                }
                low.getValue().end = Math.min(low.getValue().end, left);
            }
            Map.Entry<Integer, Interval> high = intervals.higherEntry(left);
            Interval cur = high == null ? null : high.getValue();
            while (cur != null) {
                if (cur.start >= right) break;
                intervals.remove(cur.start);
                if (right <= cur.end) {
                    cur.start = right;
                    intervals.put(cur.start, cur);
                    break;
                } else {
                    high = intervals.higherEntry(cur.start);
                    cur = high == null ? null : high.getValue();
                }
            }
        }
    }
}

/**
 * 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);
 */
```

written by leuction original link here