Data Stream as Disjoint Intervals

Given a data stream input of non-negative integers a_1, a_2, ..., a_n, ..., summarize the numbers seen so far as a list of disjoint intervals.

For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:

```
[1, 1]
[1, 1], [3, 3]
[1, 1], [3, 3], [7, 7]
[1, 3], [7, 7]
[1, 3], [6, 7]
```

Follow up:
What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?

Credits:
Special thanks to @yunhong for adding this problem and creating most of the test cases.

Solution 1

Use TreeMap to easily find the lower and higher keys, the key is the start of the interval. Merge the lower and higher intervals when necessary. The time complexity for adding is O(logN) since lowerKey(), higherKey(), put() and remove() are all O(logN). It would be O(N) if you use an ArrayList and remove an interval from it.

```
public class SummaryRanges {
    TreeMap<Integer, Interval> tree;

    public SummaryRanges() {
        tree = new TreeMap<>();
    }

    public void addNum(int val) {
        if(tree.containsKey(val)) return;
        Integer l = tree.lowerKey(val);
        Integer h = tree.higherKey(val);
        if(l != null && h != null && tree.get(l).end + 1 == val && h == val + 1)
{
            tree.get(l).end = tree.get(h).end;
            tree.remove(h);
        } else if(l != null && tree.get(l).end + 1 >= val) {
            tree.get(l).end = Math.max(tree.get(l).end, val);
        } else if(h != null && h == val + 1) {
            tree.put(val, new Interval(val, tree.get(h).end));
            tree.remove(h);
        } else {
            tree.put(val, new Interval(val, val));
        }
    }

    public List<Interval> getIntervals() {
        return new ArrayList<>(tree.values());
    }
}
```

written by qianzhige original link here

Solution 2

In general case, vector is OK, it will take O(n) time in each add, and O(1) in get result. But if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size, we'd better use another data structure "set", because the insert operation in vector will cost O(n) time, but it only cost O(log n) in binary search tree, but it will cost O(n) time in getInterval. So use which data structure will depends.

first one is the solution use vector

```
class SummaryRanges {
public:
    void addNum(int val) {
        auto Cmp = [](Interval a, Interval b) { return a.start < b.start; };
        auto it = lower_bound(vec.begin(), vec.end(), Interval(val, val), Cmp);
        int start = val, end = val;
        if(it != vec.begin() && (it-1)->end+1 >= val) it--;
        while(it != vec.end() && val+1 >= it->start && val-1 <= it->end)
        {
            start = min(start, it->start);
            end = max(end, it->end);
            it = vec.erase(it);
        }
        vec.insert(it,Interval(start, end));
    }

    vector<Interval> getIntervals() {
        return vec;
    }
private:
    vector<Interval> vec;
};
```

and below is another solution use binary search tree.

```
class SummaryRanges {
public:
    /** Initialize your data structure here. */
    void addNum(int val) {
        auto it = st.lower_bound(Interval(val, val));
        int start = val, end = val;
        if(it != st.begin() && (--it)->end+1 < val) it++;
        while(it != st.end() && val+1 >= it->start && val-1 <= it->end)
        {
            start = min(start, it->start);
            end = max(end, it->end);
            it = st.erase(it);
        }
        st.insert(it,Interval(start, end));
    }

    vector<Interval> getIntervals() {
        vector<Interval> result;
        for(auto val: st) result.push_back(val);
        return result;
    }
private:
    struct Cmp{
        bool operator()(Interval a, Interval b){ return a.start < b.start; }
    };
    set<Interval, Cmp> st;
};
```

written by goodliar original link here

Solution 3

For a new number n, find and return the index of interval [s, t] such that s is the largest 'start' that is smaller than n. If no such interval exists, return -1. This is done using binary search.

For example,

- new number 5, intervals [[1,1], [4,6], [8,8]], binary search returns 1.
- new number 0, intervals [[1,1], [4,6], [8,8]], binary search returns -1.

After we find this 'index', there are three circumstances:

1. intervals[index] already contains val. Do nothing.
2. val can be merged into intervals[index+1]. Modify intervals[index+1].start to val.
3. val can be merged into intervals[index]. Modify intervals[index].end to val.
4. val can't be merged into either interval. Insert Interval(val, val).

Finally, after inserting val, we need to check whether intervals[index] and intervals[index+1] can be merged.

```
class SummaryRanges {
private:
    vector<Interval> intervals = vector<Interval>();

    int binarySearch(vector<Interval> intervals, int val) {
        return binarySearchHelper(intervals, 0, intervals.size(), val);
    }

    int binarySearchHelper(vector<Interval> intervals, int start, int end, int va
l) {
        if (start == end) return -1;
        if (start+1 == end && intervals[start].start < val) return start;

        int mid = (start + end)/2;
        if (intervals[mid].start == val) {
            return mid;
        } else if (intervals[mid].start < val) {
            return binarySearchHelper(intervals, mid, end, val);
        } else { //intervals[mid] > val
            return binarySearchHelper(intervals, start, mid, val);
        }
    }

public:
    /** Initialize your data structure here. */
    SummaryRanges() {

    }

    /** For a new number n, find the last(biggest) interval
     *  [s,t], such that s < n. If no such interval exists,
     *  return -1
```

```
     * return -1;
     */
    void addNum(int val) {
        int index = binarySearch(intervals, val);

        // intervals[index] contains val
        if (index != -1 && intervals[index].end >= val) {
            return;
        }

        if (index != intervals.size()-1 && val + 1 == intervals[index+1].start) {
            intervals[index+1].start = val;
        } else if (index != -1 && val - 1 == intervals[index].end) {
            intervals[index].end = val;
        } else {
            intervals.insert(intervals.begin() + index + 1, Interval(val, val));
        }

        //merge intervals[index] with intervals[index+1]
        if (index != -1 && intervals[index].end + 1 == intervals[index+1].start)
{
            intervals[index].end = intervals[index+1].end;
            intervals.erase(intervals.begin()+index+1);
        }

        return;
    }

    vector<Interval> getIntervals() {
        return this->intervals;
    }
};
```

written by stellalai original link here