All O`one Data Structure

Implement a data structure supporting the following operations:

1. Inc(Key) - Inserts a new key with value 1. Or increments an existing key by 1. Key is guaranteed to be a **non-empty** string.
2. Dec(Key) - If Key's value is 1, remove it from the data structure. Otherwise decrements an existing key by 1. If the key does not exist, this function does nothing. Key is guaranteed to be a **non-empty** string.
3. GetMaxKey() - Returns one of the keys with maximal value. If no element exists, return an empty string `""`.
4. GetMinKey() - Returns one of the keys with minimal value. If no element exists, return an empty string `""`.

Challenge: Perform all these in O(1) time complexity.

Solution 1

The main idea is to maintain an ordered two-dimensional doubly-linked list (let's call it matrix for convenience), of which each row is corresponding to a value and all of the keys in the same row have the same value.

Suppose we get the following key-value pairs after some increment operations. ("A": 4 means "A" is increased four times so its value is 4, and so on.)

```
"A": 4, "B": 4, "C": 2, "D": 1
```

Then one possible matrix may look like this:

```
row0: val = 4, strs = {"A", "B"}
row1: val = 2, strs = {"C"}
row2: val = 1, strs = {"D"}
```

If we can guarantee the rows are in descending order in terms of value, then GetMaxKey()/GetMinKey() will be easy to implement in O(1) time complexity. Because the first key in the first row will always has the maximal value, and the first key in the last row will always has the minimal value.

Once a key is increased, we move the key from current row to last row if last_row.val = current_row.val + 1. Otherwise, we insert a new row before current row with vallue current_row.val + 1, and move the key to to the new row. The logic of decrement operation is similar. Obviously, by doing this, the rows will keep its descending order.

For example, after Inc("D"), the matrix will become

```
row0: val = 4, strs = {"A", "B"}
row1: val = 2, strs = {"C", "D"}
```

Inc("D") again

```
row0: val = 4, strs = {"A", "B"}
row1: val = 3, strs = {"D"}
row2: val = 2, strs = {"C"}
```

Now the key problem is how to maintain the matrix in O(1) runtime when increase/decrease a key by 1.

The answer is hash map. By using a hash map to track the position of a key in the matrix, we can access a key in the matrix in O(1). And since we use linked list to store the matrix, thus insert/move operations will all be O(1).

The psudocode of Inc() is as follows(Dec() is similar).

```
if the key isn't in the matrix:
    if the matrix is empty or the value of the last row isn't 1:
        insert a new row with value 1 to the end of the matrix, and put the key i
n the new row;
    else:
        put the key in the last row of the matrix;
else:
    if the key is at the first row or last_row.value != current_row.value + 1:
        insert a new row before current row, with value current_row.value + 1, an
d move the key to the new row;
    else:
        move the key from current row to last row;
```

Here is the code.

```
class AllOne {
public:
    struct Row {
        list<string> strs;
        int val;
        Row(const string &s, int x) : strs({s}), val(x) {}
    };

    unordered_map<string, pair<list<Row>::iterator, list<string>::iterator>> strm
ap;
    list<Row> matrix;

    /** Initialize your data structure here. */
    AllOne() {

    }

    /** Inserts a new key <Key> with value 1. Or increments an existing key by 1.
*/
    void inc(string key) {
        if (strmap.find(key) == strmap.end()) {
            if (matrix.empty() || matrix.back().val != 1) {
                auto newrow = matrix.emplace(matrix.end(), key, 1);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            else {
                auto newrow = --matrix.end();
                newrow->strs.push_front(key);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
        }
        else {
            auto row = strmap[key].first;
            auto col = strmap[key].second;
            auto lastrow = row;
            --lastrow;
            if (lastrow == matrix.end() || lastrow->val != row->val + 1) {
                auto newrow = matrix.emplace(row, key, row->val + 1);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
```

```
            else {
                auto newrow = lastrow;
                newrow->strs.push_front(key);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            row->strs.erase(col);
            if (row->strs.empty()) matrix.erase(row);
        }
    }

    /** Decrements an existing key by 1. If Key's value is 1, remove it from the d
ata structure. */
    void dec(string key) {
        if (strmap.find(key) == strmap.end()) {
            return;
        }
        else {
            auto row = strmap[key].first;
            auto col = strmap[key].second;
            if (row->val == 1) {
                row->strs.erase(col);
                if (row->strs.empty()) matrix.erase(row);
                strmap.erase(key);
                return;
            }
            auto nextrow = row;
            ++nextrow;
            if (nextrow == matrix.end() || nextrow->val != row->val - 1) {
                auto newrow = matrix.emplace(nextrow, key, row->val - 1);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            else {
                auto newrow = nextrow;
                newrow->strs.push_front(key);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            row->strs.erase(col);
            if (row->strs.empty()) matrix.erase(row);
        }
    }

    /** Returns one of the keys with maximal value. */
    string getMaxKey() {
        return matrix.empty() ?  "" : matrix.front().strs.front();
    }

    /** Returns one of the keys with Minimal value. */
    string getMinKey() {
        return matrix.empty() ?  "" : matrix.back().strs.front();
    }
};
```

written by ivancjw original link here

Solution 2

For each value, I have a bucket with all keys which have that value. The buckets are in a list, sorted by value. That allows constant time insertion/erasure and iteration to the next higher/lower value bucket. A bucket stores its keys in a hash set for easy constant time insertion/erasure/check (see first two posts here if you're worried). I also have one hash map to look up which bucket a given key is in.

Based on a previously flawed Python attempt (I just couldn't find a good way to get an arbitrary element from a set) but also influenced by an earlier version of @Ren.W's solution. We ended up with quite similiar code, I guess there's not much room for creativity once you decide on the data types to hold the data.

```
class AllOne {
public:

    void inc(string key) {

        // If the key doesn't exist, insert it with value 0.
        if (!bucketOfKey.count(key))
            bucketOfKey[key] = buckets.insert(buckets.begin(), {0, {key}});

        // Insert the key in next bucket and update the lookup.
        auto next = bucketOfKey[key], bucket = next++;
        if (next == buckets.end() || next->value > bucket->value + 1)
            next = buckets.insert(next, {bucket->value + 1, {}});
        next->keys.insert(key);
        bucketOfKey[key] = next;

        // Remove the key from its old bucket.
        bucket->keys.erase(key);
        if (bucket->keys.empty())
            buckets.erase(bucket);
    }

    void dec(string key) {

        // If the key doesn't exist, just leave.
        if (!bucketOfKey.count(key))
            return;

        // Maybe insert the key in previous bucket and update the lookup.
        auto prev = bucketOfKey[key], bucket = prev--;
        bucketOfKey.erase(key);
        if (bucket->value > 1) {
            if (bucket == buckets.begin() || prev->value < bucket->value - 1)
                prev = buckets.insert(bucket, {bucket->value - 1, {}});
            prev->keys.insert(key);
            bucketOfKey[key] = prev;
        }

        // Remove the key from its old bucket.
        bucket->keys.erase(key);
        if (bucket->keys.empty())
            buckets.erase(bucket);
```

```
    }

    string getMaxKey() {
        return buckets.empty() ? "" : *(buckets.rbegin()->keys.begin());
    }

    string getMinKey() {
        return buckets.empty() ? "" : *(buckets.begin()->keys.begin());
    }

private:
    struct Bucket { int value; unordered_set<string> keys; };
    list<Bucket> buckets;
    unordered_map<string, list<Bucket>::iterator> bucketOfKey;
};
```

written by StefanPochmann original link here

Solution 3

Main idea is to maintain a list of Bucket's, each Bucket contains all keys with the same count.

1. `head` and `tail` can ensure both `getMaxKey()` and `getMaxKey()` be done in O(1).
2. `keyCountMap` maintains the count of keys, `countBucketMap` provides O(1) access to a specific Bucket with given count. Deleting and adding a Bucket in the Bucket list cost O(1), so both `inc()` and `dec()` take strict O(1) time.

```
public class AllOne {
    // maintain a doubly linked list of Buckets
    private Bucket head;
    private Bucket tail;
    // for accessing a specific Bucket among the Bucket list in O(1) time
    private Map<Integer, Bucket> countBucketMap;
    // keep track of count of keys
    private Map<String, Integer> keyCountMap;

    // each Bucket contains all the keys with the same count
    private class Bucket {
        int count;
        Set<String> keySet;
        Bucket next;
        Bucket pre;
        public Bucket(int cnt) {
            count = cnt;
            keySet = new HashSet<>();
        }
    }

    /** Initialize your data structure here. */
    public AllOne() {
        head = new Bucket(Integer.MIN_VALUE);
        tail = new Bucket(Integer.MAX_VALUE);
        head.next = tail;
        tail.pre = head;
        countBucketMap = new HashMap<>();
        keyCountMap = new HashMap<>();
    }

    /** Inserts a new key <Key> with value 1. Or increments an existing key by 1.
*/
    public void inc(String key) {
        if (keyCountMap.containsKey(key)) {
            changeKey(key, 1);
        } else {
            keyCountMap.put(key, 1);
            if (head.next.count != 1)
                addBucketAfter(new Bucket(1), head);
            head.next.keySet.add(key);
            countBucketMap.put(1, head.next);
        }
    }
```

```
    /** Decrements an existing key by 1. If Key's value is 1, remove it from the d
ata structure. */
    public void dec(String key) {
        if (keyCountMap.containsKey(key)) {
            int count = keyCountMap.get(key);
            if (count == 1) {
                keyCountMap.remove(key);
                removeKeyFromBucket(countBucketMap.get(count), key);
            } else {
                changeKey(key, -1);
            }
        }
    }

    /** Returns one of the keys with maximal value. */
    public String getMaxKey() {
        return tail.pre == head ? "" : (String) tail.pre.keySet.iterator().next()
;
    }

    /** Returns one of the keys with Minimal value. */
    public String getMinKey() {
        return head.next == tail ? "" : (String) head.next.keySet.iterator().next
();
    }

    // helper function to make change on given key according to offset
    private void changeKey(String key, int offset) {
        int count = keyCountMap.get(key);
        keyCountMap.put(key, count + offset);
        Bucket curBucket = countBucketMap.get(count);
        Bucket newBucket;
        if (countBucketMap.containsKey(count + offset)) {
            // target Bucket already exists
            newBucket = countBucketMap.get(count + offset);
        } else {
            // add new Bucket
            newBucket = new Bucket(count + offset);
            countBucketMap.put(count + offset, newBucket);
            addBucketAfter(newBucket, offset == 1 ? curBucket : curBucket.pre);
        }
        newBucket.keySet.add(key);
        removeKeyFromBucket(curBucket, key);
    }

    private void removeKeyFromBucket(Bucket bucket, String key) {
        bucket.keySet.remove(key);
        if (bucket.keySet.size() == 0) {
            removeBucketFromList(bucket);
            countBucketMap.remove(bucket.count);
        }
    }

    private void removeBucketFromList(Bucket bucket) {
        bucket.pre.next = bucket.next;
        bucket.next.pre = bucket.pre;
```

```
        bucket.next = null;
        bucket.pre = null;
    }

    // add newBucket after preBucket
    private void addBucketAfter(Bucket newBucket, Bucket preBucket) {
        newBucket.pre = preBucket;
        newBucket.next = preBucket.next;
        preBucket.next.pre = newBucket;
        preBucket.next = newBucket;
    }
}
```

written by AaronLin1992 original link here

From Leetcoder.