There are two solutions that I know of. One I'll call "my" solution, although I think many people have come up with it. The other is the "folklore" solution that you can find if you google for this problem. Both use SegTrees, and are used in the framework of a sweepline algorithm.

1 My Solution

As usual, the SegTree implicitly represents an array of values, call them `A[]` indexed from 0 to $n-1$. In this case `A[y]` is the number of times that a given `y` value is covered by rectangles.

In each node of the SegTree we maintain three fields:

> The `delta[]` field represents a constant implicitly added to all `A[]` the values in the subtree rooted here.
>
> The `minval[]` field of a node stores the minimum value of the `A[]` in the subtree rooted there. This does not incorporate the adjustments by the `deltas` in the nodes strictly above this one.
>
> The `minvalCount[]` stores the number of leaves below this node whose current `A[]` value is equal to the `minval` of this node. (This is of course not taking the `delta[]` values strictly above this node.)

We'll see below how we can maintain these fields under the updates of inserting and deleting rectangles as we do the sweepline algorithm. But first I'll explain how this information can be used to determine the amount of the current vertical sweepline is coverered by rectangles.

Looking at the root node (node 1) of the SegTree if the `minval[1]` is zero, then the amount of length covered by rectangles is $n -$ `minvalCount[1]` (where n is the number of user variables). If `minval[1]` is non-zero, then everything in the entire range from $[0, n-1]$ is covered, so the amount covered is n.

It remains to show how to maintain the `delta[]`, `minval[]`, and `minvalCount[]` fields under updates. The updates take the form of `incrementRange(i,j,d)` which add `d` to all the elements of `A[]` between `i` and `j`.

This is done via some small modifications to the segtree code that was presented in class. Here's the key part of the code:

```
int f (int v, int l, int r, int i, int j) {
    /* We're currently at A[v].  1 <= v < 2*N.
       The range [l,r] is that of the current block, wrt user variables [0,n-1].
       The range [i,j] is the range of the query, wrt user variables [0,n-1].
       The size of the range [l,r] = r-l+1 is a power of 2.
       The range [l,r] contains the range [i,j].
       This function returns the answer to the query.
     */
    int t1, t2, m;
    if (l==i && r==j) {
        return A[v];       /* the BASE CASE part */
    } else {
        m = (l+r)/2; /* split [l,r] into [l,m] [m+1,r] */
        t1 = (i <= m)? f (LeftChild(v), l, m, i, (min(m,j))): identity;
        t2 = (j >  m)? f (RightChild(v), (m+1), r, (max(i,(m+1))), j): identity;
        return (t1 + t2);  /* the GLUE part */
    }
}

int RangeSum(int i, int j) {
    return f (1, 0, (N-1), i, j);
}
```

The changes we're going to make to handle increment range are (1) to add a new parameter `d` to the function `f()`, (2) to update the `BASE CASE` part and (3) to update the `GLUE` part. The parameter `d` is how much to add to all the values in the given range. It could be a negative number, which allows subtraction from the range.

Here's the modified code:

```
void f (int v, int l, int r, int i, int j, int d) {
    int m;
    if (l==i && r==j) {
        minval[v] = minval[v] + d;
        delta[v] = delta[v] + d;
    } else {
        m = (l+r)/2; /* split [l,r] into [l,m] [m+1,r] */
        int lcv = LeftChild(v);
        int rcv = RightChild(v);
        if (i <= m) {f (lcv, l, m, i, (min(m,j)), d)};
        if (j >  m) {f (rcv, (m+1), r, (max(i,(m+1))), j, d)};
        int mvl = minval[lcv];
        int mvr = minval[rcv];
        minval[v] = delta[v] + min(mvl, mvr);
        if (mvl < mvr)      minvalCount[v] = minvalCount[lcv];
        else if (mvl > mvr) minvalCount[v] = minvalCount[rcv];
        else                minvalCount[v] = minvalCount[lcv] + minvalCount[rcv];
    }
}
```

```
void incrementRange(int i, int j, int d) {
    f (1, 0, (N-1), i, j, d);
}

int global_non_zero_count() {
    if (minval[1] == 0) return n - minvalCount[1]; else return n;
}
```

Oh, and for the initialization, all the `delta[]` fields and `minval[]` fields are zero, and all the `minvalCount[]` fields are 1.

2 Folklore Solution

In this solution the values of `A[]` are the same as in my solution, and implicitly represented in the same way. Here we maintain just two fields in the nodes.

> The `delta[]` field, which is the same as in my solution.
>
> The `coverageLength[]` field of a node stores the number of `A[]` in the subtree rooted there whose value is > 0, without consideration of the `delta[]` values above this node.

Again, we can maintain the `coverageLength[]` fields just by modifying the `BASE CASE` and the `GLUE` parts of the code.

So the the code becomes:

```
void f (int v, int l, int r, int i, int j, int d) {
    int m;
    if (l==i && r==j) {  /* BASE case */
        delta[v] = delta[v] + d;
        if (delta[v] > 0) {
            coverageLength[v] = r-l+1;
        } else if (r > l) {
            /* delta[v]==0 and we're not at a leaf */
            coverageLength[v] = coverageLength[LeftChild(v)]
                                + coverageLength[RightChild(v)];
        } else {
            /* delta[v]=0 and we're at a leaf */
            coverageLength[v] = 0;
        }
    } else {    /* GLUE case */
        m = (l+r)/2; /* split [l,r] into [l,m] [m+1,r] */
        int lcv = LeftChild(v);
        int rcv = RightChild(v);
        if (i <= m) {f (lcv, l, m, i, (min(m,j)), d)};
        if (j >  m) {f (rcv, (m+1), r, (max(i,(m+1))), j, d)};
        if (delta[v] == 0) {
            coverageLength[v] = coverageLength[lcv]
                                + coverageLength[rcv];
        }
    }
}

int global_non_zero_count() {
    return coverageLength[1];
}
```

I have not compiled, run, and tested the code above. I want get this out quickly. If you find a problem let me know.

—Danny Sleator `<sleator@cs.cmu.edu>`