Lazy Propagation

Suppose now that the modification query asks to add to each element of a certain segment a[l...r] to some value p. As a second query, we will consider minimum in a range l to r.

To perform this modification query on a whole segment, you have to store at each vertex of the Segment Tree whether the corresponding segment has any pending additions or not. This allows us to make a "lazy" update: instead of changing all segments in the tree that cover the query segment, we only change some and leave others unchanged. In a sense, we are lazy and delay writing the new value to all those vertices. We can do this tedious task later if this is necessary.

So after the modification query is executed, some parts of the tree become irrelevant - some modifications remain unfulfilled in it.

For example if a modification query, "add a number to the whole array a[0...n−1] gets executed, in the Segment Tree only a single change is made - the number is added in the root of the tree. The remaining segments remain unchanged, although in fact the number should be added in the whole tree.

Suppose now that the second modification query says that the first half of the array a[0...n/2] should be added with some other number. To process this query we must add to each element in the whole left child of the root vertex that number. But before we do this, we must first sort out the root vertex first. The subtlety here is that the right half of the array should still be assigned to the value of the first query, and at the moment there is no information for the right half stored.

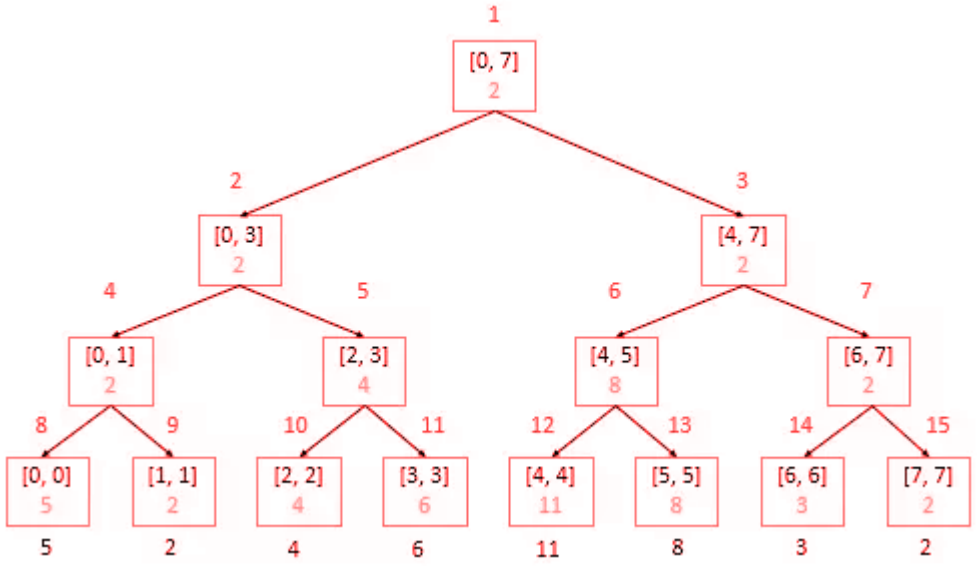
The way to solve this is to push the information of the root to its children, i.e. if the root of the tree was assigned with any number, then we add to the left and the right child vertices this number and change the minimum of the root by incrementing the minimum. We do this because adding a constant to all the values in a range increments the minimum by the same amount. A simple proof is that min(a + C, b + C) = min(a, b) + C. For other cases where we store the sum instead of the minimum, we can increase each node by (R - L + 1)*lazy[node] , where [L, R] is the range of the node. This is because each of the element contributes lazy[node] increase in sum hence a total increment of (R - L + 1)*lazy[node] takes place. Similar techniques can be used with other merge operations to propagate the changes to children.

Summarizing we get: for any queries (a modification or reading query) during the descent along the tree we should always push information from the current vertex into both of its children. We can understand this in such a way that when we descend the tree we apply delayed modifications, but exactly as much as necessary (so as not to degrade the complexity of O(logn).

For the implementation, we need to make a push function, which will receive the current vertex, and it will push the information for its vertex to both its children. We will call this function at the beginning of the query functions (but we will not call it from the leaves, because there is no need to push information from them any further).

Example

Consider the following segment tree where we are storing the minimum in a range.



Now suppose that there is a range update where we want to update the indices [2, 4] by adding 3 to all the values. So when we descend down the tree we see that node 5 is completely inside the given range. Hence we increase the lazy value of node 5 by 3 (when we increase all the values by a constant the minimum also increases by the same amount). So next time we come at node 5 or any child of this node we update the tree node value and push the updates by setting lazy[5] = 0. Similarly, the other node we need to change is node 12. But since it is already the leaf node we don't need to change the lazy value and we can directly update the value from 11 to 14.

Here is the implementation of the above idea of Lazy propagation

```
//  function to construct the tree
function construct (node, start, end, tree, lazy)
{
        if(start == end)
        {
                Leaf node will have a single element
                tree[node] = A[start]
                lazy[node] = 0
```

```
        }
        else
        {
                mid = (start + end) / 2
                //   Recurse on the left child(left child index of any node = 2*node)
                construct(2*node, start, mid, tree, lazy)
                //   Recurse on the right child(right child index of any node = 2*node + 1)
                construct(2*node+1, mid+1, end, tree, lazy)
                //   Internal node will have the minimum of both of its children
                tree[node] = min(tree[2*node] , tree[2*node+1])
                lazy[node] = 0

        }
}

//   function to push the updates to children
function push(node, tree, lazy)
{
        //   update the value at this node and push the updates to children
        tree[node] += lazy[node]
        lazy[2*node] += lazy[node]
        lazy[2*node + 1] += lazy[node]
        lazy[node] = 0
}

//   function to update the elements from l to r by adding value val to all of them
function update( node, start, end, l, r, val, tree, lazy)
{
        if(start > end)
                return
        if(start == l and end == r)
        {
                //   update lazy and return
```

```
                lazy[node] += val
                push(node, tree, lazy)
                return
        }
        else
        {
                push(node, tree, lazy)
                mid = (start + end) / 2
                update(2*node, start, end, l, min(r, mid), val, tree, lazy)
                update(2*node + 1, start, end, max(mid+1,l) r, val, tree, lazy)
                //   Internal node will have the minimum of both of its children
                tree[node] = min(tree[2*node], tree[2*node+1])
        }
}

//   function to query the minimum in range l to r
function query(node, start, end, l, r, lazy, tree)
{
        if(r < start or end < l)
        {
                //   range represented by a node is completely outside the given range
                return infinity
        }
        if(l <= start and end <= r)
        {
                //   range represented by a node is completely inside the given range
                return tree[node]
        }

        push(node, lazy, tree)
        //   range represented by a node is partially inside and partially outside the given range
        mid = (start + end) / 2
        p1 = query(2*node, start, mid, l, r, lazy, tree)
```

```
        p2 = query(2*node+1, mid+1, end, l, r, lazy, tree)
        return min(p1, p2)
}
```

Time Complexity: It is easy to see that the time complexity of all the functions - construct, update and query are the same as that in a normal segment tree - i.e O(n), O(logn), O(logn) respectively. This is due to the fact that the only addition we have made is with the push function which propagates the updates in a constant time.