CS345 Assignment 2

Chinmay Goyal(180206) and Supreeth Baliga(180801)

1 Dynamic Sequence with Rotation Operation

Solution:
We will mainly require the following tools for progressing with the algorithm:

- Special Union
- Split
- Lazy Propagation of Values

We will first define what each node of our augmented binary search tree contains. Say we are taking any node v:

- value(v) - value of the node v in the sequence
- left(v) - left child of node v
- right(v) - right child of node v
- parent(v) - parent of node v
- size(v) - number of nodes in the subtree rooted at v
- IsRotate(v) - a flag for node v (meaning explained later)

1.1 Tools:

1.1.1 Special Union:

In $SpecialUnion(T_1, T_2)$ we will provide T_1 and T_2 such that the nodes of T_1 represent lower indices than nodes of T_2. Over here we do union based on the indices that are represented by any node. Let h_1 be the height of T_1 and h_2 be the height of T_2, if $h_2 \geq h_1$, we take the leftmost node of T_2 say P, remove it, and then make the root of T_1 the left child of this node, then like the class algorithm, we find the node of T_2 with the same black height as that of T_1 and make that the right child of P. In the other case, we take the rightmost node of T_1, delete it, make the root of T_2 the right child of the node and then merge similarly as described above.
At any point of time, when we pass any node we do the LazyPropagation step as described in the Lazy Propagation section and rotation section. The time complexity of this algorithm is still *O(logn)*, we have taken the exact same algorithm of class and just added the component of LazyPropagation and the fact that here we do union based on the indices that are represented by a node. Here we should note that the binary tree is still sorted in terms of indices and an inorder traversal on such a tree would give the array that is represented by the binary tree.

1.1.2 Split:

In *Split(i,T)* (where T represents the tree we are splitting) operation in our case, we split the binary tree according to the index of the element in the dynamic sequence, let's say *i*. We find the node that represents the value of the current index , let's name the required node P and find the path leading from P to the root node of the tree (while performing LazyPropagation along the way). We maintain two trees, T_1 and T_2 where T_1 stores the nodes that represent indices less than i and T_2 store the nodes with indices greater than or equal to i.
We start from P and move towards root node, the left child of P will go in T_1 and right child along with P will go in T_2. As we move towards the root, if the node in consideration is left child of its parent, then, the parent along with right child will go in T_2 else the parent along with left child will go in T_1. Here we need to see that the node that we get in the path from P to the root, provides the node that we needed to seperately find while performing the *SpecialUnion*. Hence the time complexity of the *SpecialUnion* over here = $log(n_2) - log(n_1) = log(\frac{n_2}{n_1})$, where n_2 and $n1$ are the sizes of the two trees we are merging. As, we continue on merging trees we will get a sequence such as :

$$log(\frac{n_2}{n_1}) + log(\frac{n_3}{n_2}) + \dots\dots + log(\frac{cn}{n_t})$$

this will give us Time Complexity of *Split* to be = $O(log(n))$
Here c represents some fractional value, when we reach the root node then, if we take either the left or right child and their subtree then that subtree will have *O(n)* number of nodes, hence represented here by *cn*. After performing the *SpecialUnion*, we move to the parent of the node and repeat the same steps till we reach the root of the tree.

1.1.3 Lazy Propagation

Here, we will define lazy propagation we are using for this algorithm. So let us assume that we have to perform some operation on all the nodes in the subtree of a given node. For this, let us store a flag at each node. Initially, the flag for all the nodes is 0. Now, let us suppose we have to perform the operation on all the nodes in the subtree rooted at node x. We traverse the tree to find x. After this, we toggle the value of flag at x. (if(flag(x)==0) flag(x)=1; else flag(x)=0);
Now, the only thing we need to take care about while doing operations on this tree is that, while traversing the tree, we have to lazily propagate the flag of the nodes on the path we are traversing. Below is the algorithm we apply while traversing the tree.

```
Say we are at node v and flag(v)==1:
    Perform the operation on v;
    toggle flag of v;
    if(left(v)!=NULL) toggle flag of left(v);
    if(right(v)!=NULL) toggle flag of right(v);
After doing this, we go on to the next node in
the required path and again check for the same conditions.
```

This way, we are lazily propagating the value of flag. Meaning, we perform the operation only when the node is encountered. This helps in improving the time complexity.

Since, we have already discussed insertion, deletion and report in class, we will first focus on the rotation part. After that, we will mention the changes we have to make in the insertion, deletion and report algorithms in order to make this data structure work successfully.

1.2 Rotation

Let us break the problem into a simpler subproblem, that is rotating the complete array that is represented as an augmented binary tree as done in class. At every node of the augmented binary tree, we store a flag called *IsRotate* which value is true if we have to rotate the subtree rooted at that node. To rotate the complete binary tree, we toggle the value in its root node. Now if while traversing the tree, we encounter a node that has *IsRotate* to be true then we swap its left and right child and toggle the value of *IsRotate* in the children nodes. If we recursively rotate all the nodes then the time complexity will come out to be *O(n)* which is more than what we need to do. Hence we use Lazy Propagation as described above and perform the rotation at any node only when we need to.

1.2.1 Proof of this Simple rotation

Say the root node of the current tree is element P and represents index x(1 based indexing) which implies there are *x - 1* elements in the left subtree of this node and say y elements in the right subtree of this node(total elements in the tree = x+ y). After rotating the array, the element P will have y elements at lower indices and *x-1* elements at higher indices. Hence we swap the left and right child of *P*. Now P is at its correct position according to the rotated array, and we need to rotate the subarray represented by the right and left child of *P*. We follow the recursive approach and continue the same procedure on the right child and the left child. Now if we rotate the whole tree recursively at once, then the time complexity will be *O(n)* but we need *O(logn)*, hence we use lazy propagation to achieve this as described above.

Below is the pseudo-code for the lazy propagation operation:

```
Say while traversing the path in a tree, we are at a node v.
Let us denote the operation to be performed as LazyProp(node v):

LazyProp(node v) {
    if(IsRotate(v)==true) {
        swap(left(v),right(v));
        toggle(IsRotate(v));
        if(left(v)!=NULL) toggle(IsRotate(left(v)));
        if(right(v)!=NULL) toggle(IsRotate(right(v)));
    }
}

Once the operation is performed for v, we move on to the next node
in the required path of traversal.
```

Now comes the bigger problem of rotating only a subarray of a given array, let the subarray be from index i to index j (both included). We need to rotate from *S[i]* to *S[j]*. We know how to rotate a complete array, hence we try to modify our requirements to that simpler problem. We perform *Split(i,T)* on the tree, which splits the binary tree T into two trees T_1 and T_{temp}, where T_1 has the nodes that represents the indices less than i and T_{temp} has the nodes that represent the indices greater than equal to i.
We perform one more split *Split(j+1,*T_{temp}*)* on T_{temp}. Now, T_{temp} is further split into two trees T_2 and T_3, where T_2 has nodes that represent indices less than equal to j and T_3 has nodes that represent indices greater than j.
Now, we have three binary trees T_1, T_2 and T_3 and we have to rotate T_2, which represent the subarray *S[i] - S[j]*. We have to rotate this subarray and we know how to rotate it from

the above algorithm (NOTE: We use Lazy Propagation so we will only toggle the value of *IsRotate* in the root node of T_2). Now, we need to get our original binary tree, for this we perform merge operations: T_{temp} = SpecialUnion(T_1,T_2) and T = SpecialUnion(T_{temp}, T_3). The tree T is our binary tree which represents the array *S*. Now when we need to perform any of the mentioned 4 queries we will need to find a node, hence we will perform LazyPropagation from root to that node, hence resulting in rotation of the subarray whenever needed.

When we invoke *SpecialUnion*, LazyPropagation takes care of the flag value *IsRotate* of the nodes that our algorithm passes through, hence maintaining the correctness of our algorithm, a more detailed case wise approach is mentioned :
When we perform *SpecialUnion(T_1, T_2)*, then our tree T_2, might have less than equal height than T_1 or more height than T_1.

1. In the first case, T_2 will be added to some node of T_1 and then there may be a condition of color imbalance. In color imbalance, either the color imbalance may be directly propagated upwards, without changing the node structure of the node where T_2 was attached, hence in this case there will be no issue of rotation of the subtree T_2
   In case a rotation is performed, then also the complete subtree T_2 will be together and only the parent of the root node of the subtree T_2 might be changed, hence our flag *IsRotate* is not affected.

2. In the second case, we will attach T_1 to some node of T_2, in that case when we propagate down the left branch of T_2, then we do perform the LazyPropagation operations, hence attaching T_1 to T_2 does valid rotations on subtrees of T_2 whenever required.

1.2.2 Pseudo Code

Below is the pseudo code for the complete rotation operation.

```
Rotate(S,i,j) {
    // Keep in mind that we are performing LazyProp() even while
    traversing the paths in the Split and SpecialUnion Operations
    $T_1, T_{temp}$ <- Split(i,T);
    $T_2, T_3$ <- Split(j+1,$T_{temp}$);
    toggle(IsRotate(T2.root));
    $T_{temp}$ <- SpecialUnion($T_1, T_2$);
    T <- SpecialUnion($T_{temp}, T_3$);
}
```

1.2.3 Time Complexity Analysis of Rotation

For rotation, the operations that we had to perform are: 2* *Split* + 2**SpecialUnion* and the time complexity of both the operations is *O(logn)*. Hence the total time complexity of Rotation is *O(logn)*.

1.3 Insert

$Insert(S, i, x)$ will be same as the one discussed in class. We basically start from the root node. We check the size of the left subtree. If our required index to be placed at is less than size(left(v))+1 then we recursively have to insert the node in the i_{th} index in left subtree. Else, we have to insert the node in the $(i - 1 - size(left(v)))_{th}$ index in the right subtree. Note that, we have to increase the sizes of the nodes encountered along this path by 1 because we are inserting a new node in their subtree. The only modification is that, while traversing the tree, we will constantly be checking the value of IsRotate flag for the nodes encountered while traversal. If IsRotate is true, we will perform the LazyProp() operation as explained in the rotation subsection. Time Complexity is still *O(logn)*, as discussed in class because the at any node, Lazy Propagation step is performed in *O(1)* time and we are only traversing the height of the tree which is *O(logn)*. Note that, suppose we are doing rotation of nodes for rectifying color imbalance in the red black tree, it won't cause any problems. Because we are implementing Lazy Propagation whenever we encounter any node along the path.

1.4 Delete

$Delete(S, i)$ will be same as the one discussed in class. The algorithm for traversing the tree is similar to the one explained in insertion. Again, while traversing the tree, we will constantly be checking the value of *IsRotate* flag for the nodes encountered while traversal. If IsRotate is true, we will perform the lazy propogation operation as explained in the rotation subsection. Once the node is deleted, we need the update the value of size for all the nodes in the path from the parent of deleted node to the root. This is the same algorithm as that done in class with the only added component being that of LazyPropagation, which at any node only takes *O(1) time*. Hence time complexity of this operation is *O(logn)*. Again the color imbalance will be handled as explained in the insertion case.

1.5 Report

$Report(S, i)$ again will be same as the one discussed in class. Again, while traversing the tree, we will constantly be checking the value of *IsRotate* flag for the nodes encountered while traversal. If *IsRotate* is true, we will perform the lazy propagation operation as explained in the rotation subsection. Time complexity is still *O(logn)* (same explanation as done insert and delete).