Assignment 1

Aryan Choudhary - 170152
Abhinav Sharma - 180017

Dynamic Sequence with Rotation operation

We will be using Red-Black tree for the given operations.

Structure of each node in BST

Each node in our BST will contain following fields:

1. left - pointer to its left child
2. right - pointer to its right child
3. parent - pointer to parent node
4. value - information this node should maintain corresponding to element present at its index in the dynamic sequence.
5. color - colour of the node (0 for black and 1 for red)
6. height - represents the black height of the node i.e. no of nodes with color black on any path from this node to leaf.
7. size - number of nodes in the subtree rooted at this node
8. reverse - a boolean value whose value is 1 if the subtree rooted at this node needs to be reversed else it is 0

We explicitly define an empty tree denoted using NULL as tree with size 0, black height 0. color Black, reverse False, NULL left and right subtrees.

JOIN and SPLIT

Two major operations that we will be using to carry out various other operations are JOIN and SPLIT. In both of them will use PUSH operations which will complete all lazy operations before we touch any of its subtree.

PUSH(T) \\Auxiliary function to complete all lazy operations due at root T.

1. if **T.reverse** = **False** \\Base Case. No pending operations.
2. return
3. end if
4. SWAP(**T.left** , **T.right**)
5. FLIP **T.left.reverse**
6. FLIP **T.right.reverse**

7. **T.reverse ← False**

end

It just returns at root if there is no pending operation at root. Otherwise it swaps left and right subtree. Flips reverse of left and right subtree to add a pending operation.

Time Complexity - $O(1)$. Since SWAP and FLIP takes constant amount of time.

JOIN

It inserts a node with value X in between Left and Right tree. Returns a single combined tree.
JOIN (Left, X , Right)

1. if Left.height < Right.height
2. return JOINRIGHT(Left,X,Right)
3. else if Left.height > Right.height
4. return JOINLEFT(Left,X,Right)
5. else
6. T ← NewNode(X,Left,Right,Black)
7. \\New black colored node with value X and Left as Left subtree and Right as Right subtree. Reverse field False.
8. Return T
9. end if

end

JOINLEFT assumes that black height of left tree is strictly greater than right tree. JOINRIGHT is called when black height of right tree is strictly greater than left tree. We will provide pseudo code of JOINLEFT. JIONRIGHT is similar.

JOINLEFT (Left, X , Right)

1. while Left.height > Right.height
\\Loop right side until you reach a node with same black height as right.
2. PUSH(Left)
3. Left ← Left.Right
4. end while
5. if Left.color = Red
6. PUSH(Left)
7. Left ← Left.Right
8. end if

9. Parent ← Left.Parent \\Parent isnt NULL as above loop execute atleast once. Height is strictly more than other.

10. T ← NewNode(X,Left,Right,Red)
\\New red colored node with value X and Left as Left subtree and Right as Right subtree. Reverse field False.

11. T.Parent ← Parent

12. Parent.Right ← T

13. RB_INSERT_FIXUP(T)

end

This is the only place in (JOINLEFT or JOINRIGHT) where we will call RB_INSERT_FIXUP(T). One thing to note here is that all the ancestors of T have reverse field false. Also color of T is black.

RB_INSERT_FIXUP (T)

1. while T.parent.color = red

2. if T.parent = T.parent.parent.left
\\Parent of T is left child of its parent

3. uncle ← T.parent.parent.right

4. **PUSH(uncle)**
\\we need to call PUSH for uncle as it is not an ancestor of T. All the ancestors of T are already relaxed in JOINLEFT/JOINRIGHT procedure

5. if uncle.color = red

6. T.parent.color ← black

7. uncle.color ← black

8. T.parent.parent.color ← red

9. T ← T.parent.parent

10. else
\\color of uncle of T is black

11. if T = T.parent.right

12. T ← T.parent

13. LEFT_ROTATE(T)

14. end if

15. T.parent.color ← black

16. T.parent.parent.color ← red

17. ROTATE_RIGHT(T)

18. end if

19. else
\\similar case if parent of T is left child of its parent

20. end if

21. Update all augmented fields whose formulas are mentioned in section later uptil root.

22. if root of tree of dynamic sequence is Red. Make it Black.

end

Only changes we have made in the pseudo code is the places where we should call PUSH function compared to CLRS pseudo codes.

Rotations

LEFTROTATE and RIGHTROTATE procedures are called in RB_INSERT_FIXUP procedure. LEFTROTATE is described here and RIGHTROTATE is also similar to it.

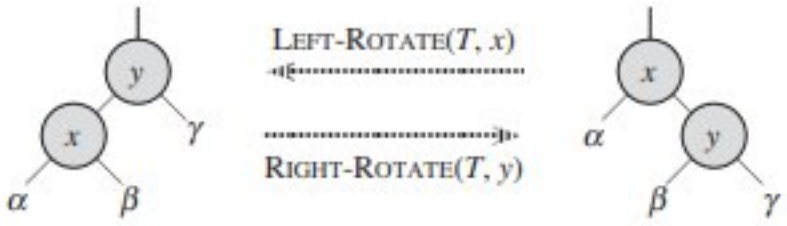


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

Image taken from CLRS

Modified CLRS pseudo code of **LEFTROTATE**.

LEFTROTATE (x)

1. y ← x.right

2. **PUSH(x.left)**

3. **PUSH(y.left)**

4. **PUSH(y.right)**

5. x.right ← y.left

```
6.      if y.left != NULL
7.          y.left.parent ← x
8.      end if
9.      y.parent ← x.parent
10.     if x.parent != NULL
11.         if x = x.parent.left
12.             x.parent.left ← y
13.         else
14.             x.parent.right ← y
15.         end if
16.     end if
17.     y.left ← x
18.     x.parent ← y
19.     update height and size fields of x and y using the recursive formula mentioned later.
end
```

Complexity Analysis of JOIN

If black height of left subtree(Left) is equal to that of right subtree(Right) , JOIN procedure executes step 6 and takes O(1) time.
Otherwise, without loss of generality, lets assume that black height of left subtree(h1) is greater than that of right subtree(h2), (i.e. h1 > h2). In this case , JOIN procedure will execute if condition in step 3 which takes O(1) time. In the step 4 JOINLEFT will be called, for which we will prove that it takes O(h1-h2) time to exectue.
Lets suppose we have a pointer at root node of L (i.e. at height h1) and it follows the right child of root node and descends one height below (i.e. at height = h1-1). Everytime this pointer follows the right child , it descends one height below the current height and also calls PUSH on current node, until it reaches a node v which is at a height h2. So, in total this pointer descends (h1-h2) black height or we can say that steps 1-3 of the JOINLEFT procedure runs until the pointer passes through h1-h2 black nodes. This means, steps 2-3 runs for at max $2 * (h1 - h2)$ times (since, there can be at max one red node between two black nodes) and step 1 runs for at max $2 * (h1 - h2) + 1$ times. Steps 5-12 take constant time.

RB_INSERT_FIXUP is executed in order to do away with any color imbalance that may appear while inserting the right subtree at height h2. In this procedure the color imbalance travels up and fix the red-black tree property in the subtree rooted at each of these nodes. Each steps 2-18 will be performed in O(1) time, as the time complexity for rotation operation is also O(1). In each execution of while loop in step 1, T is either set to its parent (in case of Rotation) or its grandparent, so in worst case step 1 will be executed for number of times which is equal to initial depth of T from the root of tree. Or, we can say that step 1 will be executed for at max (h1-h2) times , where h1 is the total height of tree and h2 is the height at which node T is initially present.
Therefore, complexity of step 13 in JOINLEFT is O(distance of node at which right subtree is inserted), i.e. c1*(h1-h2).

Overall complexity of JOIN procedure is **O(h1-h2)**. In general, time complexity is **O(max(h1,h2)-min(h1,h2))**, where max(h1,h2) is the larger of the heights of trees which are to be joined and min(h1,h2) is the smaller of the

two.
We can also say, time complexity is **O(max(H1,H2)-min(H1,H2))**, where H1, H2 are resp. black heights. Since overall height is proportional to black height.

Now, lets see what happens to the black height of the combined tree with respect to that of each of the two subtrees.
In case , where black height of two subtree are same, we simply join them using the given node as root and the two given tree as its children and color the root as black. Therefore, the black height of root is black height of its child+1.
Otherwise, without loss of generality, lets assume that black height of left subtree(h1) is greater than that of right subtree(h2), (i.e. h1 > h2). In this case, when we descend down the left subtree, the black height of its root remains same. This height may only change when , there is some property violation of red-black tree and rotation need to be performed at the root. Even in this case, due to just one rotation at the root node, the black height may change by at most 1 (when we just change root's color from red to black).
So, we can say that, after merging two red-black tree, the black height of the resultant tree changes by atmost one.

SPLIT

SPLIT(T,X) Its divides the given tree into 2 trees namely left tree and right tree such that left tree has exactly first X nodes from inorder traversal of T. It assumes that given tree as atleast X nodes. It also maintains inorder traversal i.e. writing inorder traversal of left tree first and then right tree will be same as writing inorder traversal of given tree.

SPLIT(T,X)

1. if **T.size = X** \\Base Case
2. return **T** , **NULL**
3. else if **X = 0**
4. return **NULL** , **T**
5. end if
6.
7. **PUSH(T)**
8. **Left_Subtree** , **Root** , **Right_Subtree** ← **SPLIT_ROOT**(T)
9. **Root_Rank** ← **1+Left.size**
10.
11. if **Root_Rank** ≤ **X**
12. **Left_Tree** , **Right_Tree** ← **SPLIT(Right,X - Root_Rank)**
13. return **Join(Left_Subtree,Root,Left_Tree)** , **Right_Tree**
14. else
15. **Left_Tree** , **Right_Tree** ← **SPLIT(Left,X)**
16. return **Left_Tree** , **Join(Right_Tree,Root,Right_Subtree)** ,
17. end if

end

This function calls SPLIT_ROOT(T) and PUSH(T). PUSH(T) first completes pending operations on root if any. Then SPLIT_ROOT(T) splits given tree at root and gives us value at root (denoted by Root), left and right subtree of root. It also fixes parent pointers of left and right. If root of left subtree (and/or right subtree) is red it sets colors of root to black and increases black height of that tree.

Correctness of SPLIT

If X is equal to 0. Then we are done. We can just return an empty tree as left tree and complete T as right tree.
If X is equal to the size of tree then also we are done. Just return complete T as left tree and empty tree as right tree.

Otherwise we first propagate pending operation at root to its subtrees, which takes constant time.

Then we find rank of root to determine in which tree root lies.
If rank of root is less than and equal to X we know for sure root and left subtree lies in left tree. Later on recursively we find first $X - Root\_Rank$ of right subtree. And we just attach those after root. We also attach left subtree before root since they come before root in inorder traversal. This indeed contains first X nodes of tree because we know that left subtree contains atmax $X - 1$ nodes. Hence all nodes in left subtree and root will be forsure among first X nodes in inorder traversal. Proof of correctness is by induction on problem size. It finds remaining sufficient no nodes from right subtree correctly size of subtree is less than complete tree (Inductive tree). Our base cases are correct.

Other case is similar. We know that left subtree contains atleast X nodes. Hence root and right subtree cannot belong to first X nodes in inorder traversal. We can just find those X nodes from left subtree. Then to maintain correct inorder traversal of right tree. We can attach remaining nodes of left subtree first (since they come before root) then root and then right subtree.

Complexity Analysis of SPLIT

Let N be the no of nodes in tree. Let $\frac{\log N}{2} \leq H \leq 2 * \log N$ be the black height of T. We know that there are atmax $2 * H$ nodes on any path from root to leaf. Hence we call $SPLIT(T, X)$ atmax $2 * H$ times. Also statements in Line 1-9 takes $O(1)$. Hence these all statements in line 1-9 takes $O(H)$ together.

What remains to prove is that all join calls in line 13 together takes $O(H)$ time. Proof for line 16 is similar. We will leave that.

Let L_1,L_2,...,L_k be the left subtrees used to construct left tree in spilt operation i.e. first L_1 and L_2 are joined. Then their result is joined with L_3 and so on... at last they are joined with L_k as we exit recursive calls of split. Also note, $k \leq 2 * H$.
Let us denote the resultant of merging L_1,L_2,...,L_i in that order by S_i. Where S_{i+1} the tree formed by applying join operation on L_{i+1}, S_i along with mid at that relevant position.

Let black heights of L_1,L_2,...,L_k as $H_1, H_2, ..., H_k$ resp.
Let black heights of S_1,S_2,...,S_k as $HS_1, HS_2, ..., HS_k$ resp.

We also know that black height never increases as we go down the tree. Hence, $H_1 \leq H_2 \leq ... \leq H_k$
Let T_i be the time taken to form S_i.
$T_{i+1} = T_i + c * (H_{i+1} - HS_i + 1)$ where $HS_i \leq H_i + c_2$ where c_2 is delta black height increase due to joining.
$\implies T_{i+1} \leq T_i + c * (H_{i+1} - H_i + c_1)$ where $c_2 = c_1 + 1$

Where T_i is the time taken to form S_i and $c * (H_{i+1} - H_i + c_1)$ is the time taken to destroy S_i and create S_{i+1}.
Let $T_0 = 0$, since we dont create 0th tree.

We are interested in T_k time taken to from S_k and destroy all other S_i for $1 \leq i < k$.

Solving these equations -

$$T_k \leq T_{k-1} + c * (H_k - H_{k-1} + c_1)$$
$$\vdots$$
$$T_{i+1} \leq T_i + c * (H_{i+1} - H_i + c_1)$$
$$\vdots$$
$$T_2 \leq T_1 + c * (H_2 - H_1 + c_1)$$
$$T_1 \leq T_0 + c * (H_1 - H_0 + c_1)$$

Adding all these -

$T_k \leq T_0 + c * (H_k - H_0) + k * c_1 * c$
Hence,
$T_k = O(H_k)$
$\implies T_k = O(\log N)$
N is the size of S_k tree.

Informally, time taken by join operation is proportional to difference of black heights of two trees we want to join. Since black height of nodes to be merged always increases. At any time we either merge two trees of approximately same height (hence O(1)) or it increase overall maximum height. Total time taken is proportional to difference between maximum black height (i.e. H_K) and minimum black height H_1.

Recursive formulas

We will state recursive relations of all fields (except "reverse" field) of a node in terms of its own fields as well as the fields of its children. Those formulas are sufficient to show that all these fields can be maintained. We have avoided adding pseudo codes of how to maintain these fields. So that pseudo codes look small. 'reverse' is always explicitly set to False before any rotation (which denotes no pending operations). Hence, even after rotation they are False (to denote no pending operations).

left, right, parent, color are standard fields of Red Black tree. We arent augment them.
Each node has a unique value assigned to it. It remains same at all moments of time. We dont touch it during rotation.

T.size = T.left.size + T.right.size + 1
T.height = T.left.height if T.color is Red.
T.height = T.left.height + 1 if T.color is Black.

Operations

Now we will show how one can use SPLIT and JOIN operations to do all required operations.
N is the size of dynamic sequence at the time of operation.

Insert

$1 \leq i \leq N + 1$
Insert(S,i,x)

1. if **S.size+1 == i** \\Base Case. Insert in the end.
2. return **JOIN(S,x,NULL)**

3. else if **i == 1** \\Base Case. Insert in the beginning.
4. return **JOIN(NULL,x,S)**
5. end if
6. **Left , Right ← SPLIT(S,i-1)** \\Split the sequence that left tree has i-1 nodes. Add it after Left.
7. return **JOIN(Left,x,Right)**

end

Time Complexity - $O(\log N)$.
Constant no of SPLIT and JOIN operations and each takes $O(\log N)$.

Delete

$1 \leq i \leq N$
Delete(S,i)

1. if **i == 1** \\Base Case. Delete first.
2. Left , Right ← SPLIT(S,1)
3. return Right
4. else if **i == N** \\Base Case. Delete Last
5. Left , Right ← SPLIT(S,$N-1$)
6. return Left
7. end if
8. Left , Right ← SPLIT(S,$i-1$) \\Split just before ith element.
9. ithElement , Right ← SPLIT(Right,1) \\Separate ith element from others. Now forget it.
10. x , Right ← SPLIT(Right,1) \\Extract first element from right subtree. Its size it atleast 1. Use it to join with other part.
11. return JOIN(Left,x.value,Right)

end

Time Complexity - $O(\log N)$.
Constant no of SPLIT and JOIN operations and each takes $O(\log N)$.

Report

$1 \leq i \leq N$
Report(S,i)

1. Left,Right ← SPLIT(S,$i-1$) \\Split just before ith element.
2. ithElement,Right ← SPLIT(Right,1) \\Separate ith element from others.
3. value ← ithElement.value
4. T ← JOIN(Left, value, Right) \\Join sequence again.

5. return value

end

Time Complexity - $O(\log N)$.
Constant no of SPLIT and JOIN operations and each takes $O(\log N)$.

Rotate

$1 \leq i \leq j \leq N$
Rotate(S,i,j)

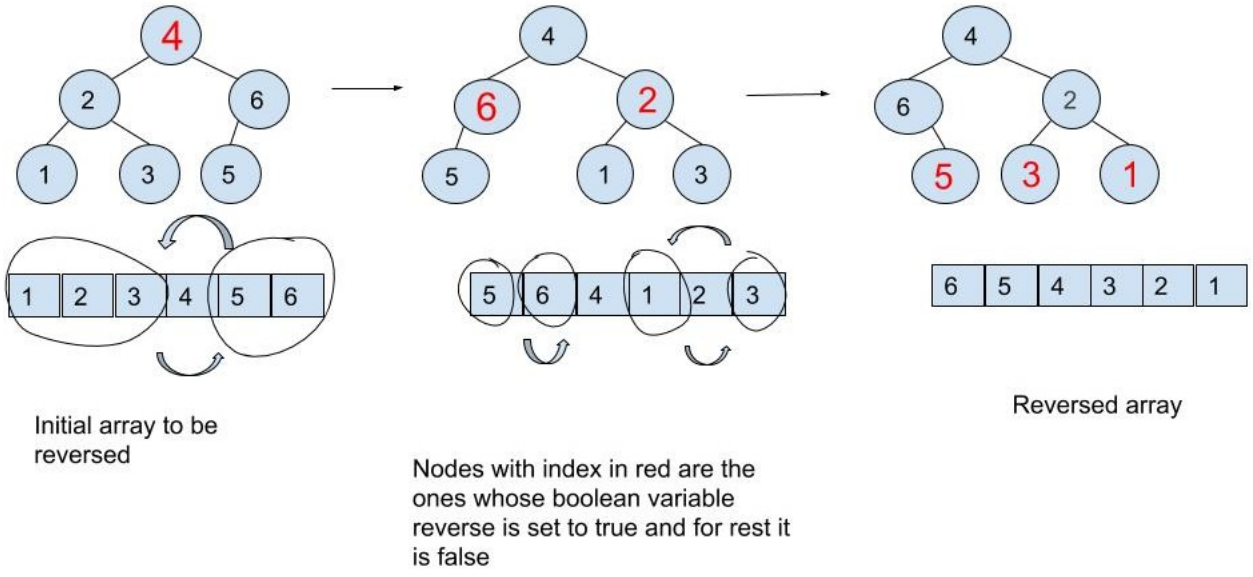
1. Middle,Right ← SPLIT(S,j) \\Separate first j elements.
2. Left,Middle ← SPLIT(Middle,i-1) \\Separate first i-1 elements.
3. FLIP Middle.reverse
4. x,T ← SPLIT(Middle,1) \\Join Left and Middle
5. T ← JOIN(Left, x.value, T)
6. T,x ← SPLIT(T,T.size - 1) \\Join Rest with right part.
7. T ← JOIN(T, x.value, Right)

end

Time Complexity - $O(\log N)$.
Constant no of SPLIT and JOIN operations and each takes $O(\log N)$.

We divide sequence into 3 parts. Left, Middle and Right. Where Middle contains all elements in range $[i, j]$. Then we reverse it by flipping the reverse variable at root of middle. After that we just join 3 parts back.

Illustration



These changes are done lazily. Only when we need are forced to go down the tree. These lazy operation is what makes all operations in $O(\log N)$