CS 345A Assignment 2

Yasharth Bajpai (170822), Nidhi Hegde (180472)

September 24, 2020

Moderate

Overview:

The idea of the data structure to be used for performing all the five operations in worst case $O(\log n)$ time, is that of an **Balanced Binary Search Tree augmented with certain fields** at every node so as to help us achieve the aforementioned complexity.

Node Structure

1. **val** : Stores the value stored at a node.
2. **left, right, parent** : Stores the left child, right child and parent of a node.
3. **size** : Stores the size of the subtree rooted at the node.
4. **incr** : Stores the increment to be added to the values of all nodes of the subtree rooted at the node.
5. **min** : Stores the minimum values of all nodes in the subtree rooted at the node.

Algorithm 1 Insert a number with value c at i^{th} place in the sequence S

```
function INSERT(S, i, c)
    function RECURSIVEINSERT(S, i, c)
        if (S == Null) then
            u = CREATENODE(c)
            return u
        else
            size(S) ⟵ size(S) + 1
            c ⟵ c − incr(S)                    ▷ Adjusting the value of the element with respect to increments
            if (left(S) == NULL) then
                s ⟵ 0
            else
                s ⟵ size(left(S))
            end if
            if (i ≤ s + 1) then
                left(S) ⟵RECURSIVEINSERT(left(S), i, c)
            else
                right(S) ⟵RECURSIVEINSERT(right(S), i − 1 − s, c)
            end if
        end if
        return S
    end function
    S = RECURSIVEINSERT(S, i, c)
    REBALANCE(GetNode(S, i))                   ▷ To do away with any imbalance post insertion
    UPDATEMIN(GetNode(S, i))                   ▷ To maintain the min field post insertion
end function
```

Algorithm 2 Delete a node at i^{th} place in the sequence S

function DELETE(S,j)
  if ($S == Null$) **then**
    return -1;
  end if

  $u \longleftarrow$ GETNODE(S, j)

  if ($left(u) == Null$) **then**
    if $left(parent(u)) == u$ **then** //join parent to its right child
      $left(parent(u)) \longleftarrow right(u)$
    else
      $right(parent(u)) \longleftarrow right(u)$
    end if
    $incr(right(u)) = incr(right(u)) + incr(u)$ //Modify *incr* field of $right(u)$ after deletion
    UPDATESIZE($parent(u)$)
    REBALANCE($parent(u)$) //leverages $MODIFIEDROTATE$ helper to re-balance the tree
    UPDATEMIN($parent(u)$)
    return S
  end if

  if ($right(u) == Null$) **then**
    if $left(parent(u)) == u$ **then** //join parent to its left child
      $left(parent(u)) \longleftarrow left(u)$
    else
      $right(parent(u)) \longleftarrow left(u)$
    end if
    $incr(left(u)) = incr(left(u)) + incr(u)$ //Modify *incr* field of $left(u)$ after deletion
    UPDATESIZE($parent(u)$)
    REBALANCE($parent(u)$)
    UPDATEMIN($parent(u)$)
    return S
  end if

  $pred \longleftarrow$ GETNODE($S, j-1$)
  SWAP(S, u, $pred$) // Swap values of nodes u and pred
  return DELETE(S,$j-1$)
end function

Algorithm 3 Report the number present at i^{th} place in the sequence S

```
function REPORT(S, i)
    found ⟵ 0
    cumIncr ⟵ 0
    while (!found) do
        if (left(S) == NULL) then
            s ⟵ 0
        else
            s ⟵ size(left(S))
        end if
        cumIncr ⟵ cumIncr + incr(S)            ▷ Adding increment fields at all nodes in path
        if (s = i − 1) then
            found = true
        else if (s > i − 1) then
            S ⟵ left(S)
        else
            i ⟵ i − s − 1
            S ⟵ right(S)
        end if
    end while
    return (val(S) + cumIncr)    ▷ Returning the value added with the cummulative increment upto the
ith element
end function
```

Algorithm 4 Minimum value in the interval i to j

```
function MIN(S, i, j)
    u ⟵ GETNODE(S, i)
    v ⟵ GETNODE(S, j)
    w = LCA(u, v)

    minLeft ⟵ val(u)      // Initialising minLeft and minRight as value of nodes u and v
    minRight ⟵ val(v)

    if (u ≠ w) then
        if right(u) ≠ NULL then
            minLeft ⟵ MINIMUMOF(minLeft, min(right(u)) + incr(right(u)))
        end if
        minLeft = minLeft + incr(u)

        while (parent(u) ≠ w) do
            if (u == left(parent(u)) then
                if (right(parent(u)) ≠ NULL) then
                    minLeft ⟵ MINIMUMOF(minLeft, min(right(parent(u))) + incr(right(parent(u))))
                end if
                minLeft ⟵ MINIMUMOF(minLeft, val(parent(u)))
            end if
            u ⟵ parent(u)
            minLeft ⟵ minLeft + incr(u)      //Add incr field of the current node to minLeft
        end while
    end if

    if (v ≠ w) then
        if left(v) ≠ NULL then
            minRight ⟵ MINIMUMOF(minRight, min(left(v)) + incr(left(v)))
        end if
        minRight = minRight + incr(v)

        while (parent(v) ≠ w) do
            if (v == right(parent(v)) then
                if (left(parent(v)) ≠ NULL) then
                    minRight ⟵ MINIMUMOF(minRight, min(left(parent(v))) + incr(left(parent(v))))
                end if
                minRight ⟵ MINIMUMOF(minRight, val(parent(v)))
            end if
            v ⟵ parent(v)
            minRight ⟵ minRight + incr(v)
        end while
    end if

    minVal ⟵ MINIMUMOF(val(w), minLeft, minRight)

    while w ≠ S do
        minVal ⟵ minVal + incr(w)      //collecting increments for node w till the root node
        w ⟵ parent(w)
    end while

    return minVal
end function
```

Algorithm 5 Add x to each number at places $i, i+1, ..., j$ in the sequence S

```
function ADD(S, i, j, x)
    u ⟵ GETNODE(S, i)
    v ⟵ GETNODE(S, j)
    w ⟵LCA(u,v)
    val(w) ⟵ val(w) + x                                          ▷ Addition to u

    if (u ≠ w) then
        val(u) ⟵ val(u) + x                                      ▷ Addition to u
        if (right(u) ≠ Null) then
            incr(right(u)) ⟵ incr(right(u)) + x
        end if
        while (parent(u) ≠ w) do                                  ▷ Traversing upto w
            if (u == left(parent(u)) then
                if (right(parent(u)) ≠ Null) then    ▷ Increment for elements in range and off path
                    incr(right(parent(u))) ⟵ incr(right(parent(u))) + x
                end if
                val(parent(u)) ⟵ val(parent(u)) + x    ▷ Addition to elements on path and in range
            end if
            u ⟵ parent(u)
        end while
        UPDATEMIN(GetNode(S, i))
    end if

    if (v ≠ w) then
        val(v) ⟵ val(v) + x                                      ▷ Addition to v
        if (left(u) ≠ Null) then
            incr(left(v)) ⟵ incr(left(v)) + x
        end if
        while (parent(v) ≠ w) do                                  ▷ Traversing upto w
            if (v == right(parent(v)) then
                if (left(parent(v)) ≠ Null) then    ▷ Increment for elements in range and off path
                    incr(left(parent(v))) ⟵ incr(left(parent(v))) + x
                end if
                val(parent(v)) ⟵ val(parent(v)) + x    ▷ Addition to elements on path and in range
            end if
            v ⟵ parent(v)
        end while
        UPDATEMIN(GetNode(S, j))          ▷ Maintain min field on all ancestors of i & j
    end if

end function
```

Algorithm 6 Helper Functions to Maintain the BST

function UPDATEMIN(u) *//updates the minimum on the path from u to root S*
  while $u \neq Null$ **do**
    if $left(u) \neq Null$ **then**
      $min(u) \longleftarrow$ MINIMUMOF($min(u)$, $min(left(u)) + incr(left(u))$)
    end if
    if $right(u) \neq Null$ **then**
      $min(u) \longleftarrow$ MINIMUMOF($min(u)$, $min(right(u)) + incr(right(u))$)
    end if
    $u \longleftarrow parent(u)$
  end while
end function

function UPDATESIZE(u) *//updates the size of tree along the path from u to root S*
  $size(u) \longleftarrow 1$
  while $u \neq Null$ **do**
    if $left(u) \neq Null$ **then**
      $size(u) \longleftarrow size(u) + size(left(u))$
    end if
    if $right(u) \neq Null$ **then**
      $size(u) \longleftarrow size(u) + size(right(u))$
    end if
    $u \longleftarrow parent(u)$
  end while
end function

function CREATENODE(c) *//creates a new node*
  Create new node u
  $val(u) \longleftarrow c$
  $left(u) \longleftarrow right(u) \longleftarrow parent(u) \longleftarrow Null$
  $incr(u) \longleftarrow 0$
  $min(u) \longleftarrow val(u)$
  return u
end function

function MODIFIEDLEFTROTATE(u) ▷ Similarly Modified Right Rotate
    $v \longleftarrow right(u)$
    $incr_u \longleftarrow incr(u)$
    if $(v \neq Null)$ **then**
        $incr_v \longleftarrow incr(v)$
        $incr(u) \longleftarrow -incr_v$
        $incr(v) \longleftarrow incr_u + incr_v$
        if $(left(v) \neq Null)$ **then**
            $incr(left(v)) \longleftarrow incr_v$
        end if
        $size(v) \longleftarrow size(u)$
        $size(u) \longleftarrow size(u) - 1$
        if $(right(v) \neq Null)$ **then**
            $size(u) \longleftarrow size(right(v))$
        end if
    end if
    LEFTROTATE(u) ▷ The Regular Left Rotate
    return u
    // REBALANCE() must UPDATEMIN( ) after the balance has been achieved
end function

---

function GETNODE(S, i)
    $found \longleftarrow false$
    while $(!found)$ **do**
        if $(left(S) == NULL)$ **then**
            $s \longleftarrow 0$
        else
            $s \longleftarrow size(left(S))$
        end if
        if $(s = i - 1)$ **then**
            $found = true$
        else if $(s > i - 1)$ **then**
            $S \longleftarrow left(S)$
        else
            $i \longleftarrow i - s - 1$
            $S \longleftarrow right(S)$
        end if
    end while
    return S
end function

---

function GETINC(u, $pred$) //returns the increment collected from node u (exclusive) to pred(inclusive)
    $inc = 0$
    while $(pred \neq u)$ **do**
        $inc \longleftarrow inc + incr(pred)$
        $pred \longleftarrow parent(pred)$
    end while
    return inc
end function

function SWAP(u, $pred$) // Swaps and modifies the *val* field of nodes u and $pred$, accomodating the change in position

$inc* = \textsc{GetInc}(u, pred)$

$temp \longleftarrow (val(u) - inc*)$

$val(u) \longleftarrow (val(pred) + inc*)$

$val(pred) \longleftarrow temp$

return

end function

Proof of Correctness

We work with a balanced binary tree, and implicitly maintain the left, right and parent pointers of all the nodes during every operation below, just as in any other balanced BST. However, we explicitly maintain the other fields *min*,*incr*,*size*, for the operations described below:

1. **Insert**
   An insert operation inserts a new node with value c at position i in the tree. We subtract from c,the increments of all the nodes on the path from root to the position of new node and finally insert the *updated* c at position i. So, when we report the value of the new node, we implicitly add all the increments along the path to the value stored at new node, which nullifies the effect of previously subtracting these increments. *Balance* and $UpdateMin$ operations are carried out after inserting node i, which re-balances the tree, and updates the *min* field of all the nodes on the path from new node to root. Thus, every associated field can be maintained during insert operation.

2. **Delete:**
   For the delete operation, we have 3 cases:

   (a) If the node has left child NULL:
   The right child of the node is simply connected to the parent of the node. The *incr* field of this deleted node is added to the *incr* field of its right child. The *incr* fields of all the other nodes need no change. *Balance* and *UpdateMin* balance the tree and update the *min* field of all the nodes in the path from deleted node to root. Thus, all the fields are balanced in this case.

   (b) If the node has right child NULL:
   The left child of the node is simply connected to the parent of the node. The *incr* field of this deleted node is added to the *incr* field of its left child. For the rest of the fields, similar argument holds for this case as (a).

   (c) If both the children are not NULL:
   Here, our idea is to swap the value of this node with its predecessor node, and then delete the predecessor node which now contains the value to be deleted. For swapping, $SWAP$ function is called which swaps the *val* field at both these nodes.
   No change is made to *incr* or *min* field at both the positions. No pointers are manipulated. Only the *val* field at position j is swapped with its predecessor $j-1$, in the tree.
   This ensures correctness for *incr* field since we are not modifying the *incr* field at any node, thus for all the nodes in the tree except j and $j-1$, *incr* field's value is still correct. Now, we need to modify the *val* field at these 2 nodes, to adapt to the change in their position after swapping. Since the node at j now contains the value of its predecessor, we need to increment its new value by a quantity *inc*$*$, which is equal to the sum of *incr* fields in the path from node at position j (exclusive) to $j-1$ (inclusive). The same quantity needs to be subtracted from the *val* field at position $j-1$, to nullify the effect of adding *incr* field twice since *Report* routine adds the accumulated *incr* fields to *val* of node, before reporting an element.

Now, for deleting the node at position $j-1$, we call DELETE$(S, j-1)$. We are ensured that this call converges to case (b) since its predecessor will have the right child $NULL$. Thus, we implicitly call $Balance$ and $UpdateMin$ in this case also, and hence maintain all the associated fields during the delete operation.

3. **Report**

   A report operation on the Sequence reports the element at i^{th} position in the sequence. To implement the Addition functionality, a field *incr* is added to each node. This reduces the time complexity of the addition operation by virtually denoting increment to the values. However, when the user needs back these values from the data structure, we must take into account these virtual increments in the value that we return while we report an index.

   The algorithm is heavily derived from the Report discussed during lectures, the only difference being the addition of increment(*incr*) fields at every node of the path in the traversal to the i^{th} node as cummulative increment(*cumIncr*) upto that node. This is then added to the value(*val*) stored at the i^{th} element to recover and report the actual value of the element back to the user.

4. **Min:**

   The *min* field at any node contains the minimum value in the sub-tree rooted at that node.

   For any 2 nodes u and v representing index i and j respectively, we need to look at all the nodes in this range. Let the least common ancestor of both the nodes u and v be LCA.

   Any node in the path from u to LCA will lie in the desired index range, if its left child lies in the path. That also marks the right subtree of this node in the desired range, as discussed in class. Now, we need to consider the minimum among this node's value and the minimum of its right subtree. In order to accomodate *Add*, we accumulate all the *incr* values in the path and add it to our $minValue$. Similarly, along the path from v to LCA, we only consider those nodes to be in range whose right child lies in the path itself.

   Once, we take the minimum along both the paths, we compare it with the *val* field of the LCA node. The minimum of all the three values, added to the accumulated *incr* field along the path from LCA to root, is our required value, since we need to take into account all the increments from root to LCA as well before reporting the min element.

5. **Add:** An Add operation added the value x to all elements from i^{th} to j^{th} index of the sequence. It is trivial to see that visiting and adding the value to every element individually is $O(n)$.

   To do better than this we must devise an implementation which helps us perform this addition without having to visit all elements in the interval. Our Add operation and tree structure is heavily derived from the one taught in the lectures. The only change being the maintenance of the minimum(*min*) field alongside the increment(*incr*) field. The Operation firstly retrieves the least common ancestor of the i^{th} and j^{th} elements(As all the elements in this interval would lie in a subtree rooted at the $w(LCA)$). We very well know that three points $u, v\&w$(which might coincide) are bound to be on the range $[i, j]$. Thus, we add the value x to these directly.

   Then we traverse up from the i^{th} element to the LCA, we realize that only the children which are the

right child of their parents on the path their parent lie inside the required range. We add x to val field of these nodes(on the path the range), whilst adding x to the right child of such node, if it exists. It is virtual addition which denotes that all in the subtree rooted at this node were incremented by x. This is levered again when the we report the elements.
Similarly, on moving from v to w, we only consider those nodes to be in range whose right child lies in the path itself. The value(val) of all such nodes on the path and in the range is added by x, while the left child of all such nodes has its $incr$ field incremented by x. Once, the addition is complete, we must maintain the min fields of the tree with the added values taken into account. This is achieved by calling $UpdateMin$ function at both i & j which settles the value of the minimum field up to the root.