CS345 Assignment 2

Raktim Mitra(150562)
Pranjal Prasoon(150508)

29 August 2017

Question 1 (Hard Version)

Data Structure

We compute an onion like shape via consecutive Convex hull computation by the following process. Steps to create the data structure needed :

- Compute the convex hull of the point set.
- Divide the convex hull into two parts 1) upper to the points with highest and lowest x-codinate (Up hull) and 2) lower to the same (Low hull). Store the points in these in two separate BSTs sorted according to x-coordinate.
- We store the points in Up hull and Low hull in two linked lists. This is done for efficient Successor and Predecessor operations.
- Remove all points on the hull and until the point set is empty repeat the process.

Algorithm

Starting from the outermost hull we compute points on upper side of the line and add them to output array, if some hull has all points on the upper side we stop and add all remaining points to output. If some hull has no point on the upper side we stop, no need to consider rest of the points as they are contained in the present hull and are not on the upper side too.

Computing intersection points of a convex hull and a line: A line can intersect a hull in two points, (excluding grazing) either both on the Up hull or Low hull or one at Up hull and one at Low hull.

Lemma: When two points (x1,y1) and (x2,y2) are substituted in the equation of the line i.e. $y = f(x)$, then the sign of $(y - f(x))$ is same if they lie on the same side of the line and it is greater than 0 if for a point lying in the upper region.

- When line cuts any half hull in one point : We do a modified binary search in which we move to the side which has lower distance from the line until we get two adjacent points with opposite sign of (y - f(x)) i.e. towards an opposite sign point and in O(log h) time we find the point of intersection (i.e. two points between which the intersection point is. Exact coordinates does not matter). We do the same in the other half hull.

- when the line cuts the same half hull in two points we use binary search to find a point on the side of the line away from the right extreme, we split the BST at this point and apply the previous step on both part of the half hull to get the intersection points.

Pseudocode: Let h_i be number of points in ith hull and k_i be number of output points on the ith hull.

Algorithm 1 Finding points on upper side of a line

```
1: procedure FIND(line L, Data Structure D)
2:     Output = [ ]
3:     for each hull from the outermost do
4:         (A,B) ← ComputeIntersection(Current hull, L)        ▷ O(log h_i) time
5:         Go from A to B in sequential order through the upper side using the
   linked list and add the points to output.                   ▷ O(k_i) time.
6:         if All points of current hull taken then
7:             Add rest all points to Output
8:             Return Output
9:         end if
10:        if no point of current hull is taken then
11:            Return Output
12:        end if
13:    end for
14:    Return Output
15: end procedure
```

Proof of Correctness

- Since convex hull of a point set is unique we will have an unique onion configuration.

- We do not exclude any point which is on the upper side of the line since we include only points on each hull if they on the upper side of line and once we get all points of a hull in upper side of line, we know all the remaining inner points are going to be on the upper side.

- similar argument shows we are not including any point lying on lower side of line .

Hence, we are reporting exactly the points that lie on the upper side of the line. [Proved]

Time Complexity Analysis

For each hull i we compute O(logh_i) to find intersection points, O(k_i) time add the correct points to output using the linked lists.

Total time T = O(logh_1) + O(k_1) + O(logh_2) + O(k_2) + ... The sum is difficult to compute, so, we focus on the worst case scenario to find the time complexity bound. The worst case is when we find only one point on every hull which is on the upper side of the line. So, number of output points = k = number of hulls examined . Clearly sum of the k_is is k. So, worst case time complexity becomes k*O(log n) + O(k) i.e. O(k log n).

Question 2 (Hard Version)

Data Structure

We maintain an augmented Red-Black tree with the following additional fields for a node **v** :

- **incr** : If $incr(v) = x$, it means that all values in $T(v)$ will be incremented by x.
- **size** : $size(v)$ gives the number of nodes in $T(v)$.
- **min** : $min(v)$ gives the number of nodes in $T(v)$.

Algorithms

• **Note** : For this particular question, as instructed by sir, we have assumed that we are totally familiar with all the basic operations in the R-B tree and for a benchmark, we have time and again referred to these algorithms as mentioned in Cormen-Leiserson (CLRS) Intro. to Algorithms.

We make use of all the basic algorithms (insert, delete, rotate) used in a Red-Black tree (specifically those in *Introduction to Algorithms, 3e : Cormen, Leiserson and others*).

While most of the algorithm remains same for the standard fields, we show how our augmented fields change.

Insert(D,i,x)

The pseudo-code is as follows:

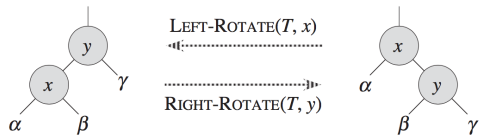
Algorithm 2 Inserting an element **x** at **i**-th position

```
1:  procedure INSERT(D,i,x)
2:      if D = NULL then
3:          val(u) ← x
4:          min(u) ← x
5:          left(u) ← NULL
6:          right(u) ← NULL
7:          size(u) ← 1
8:          return u
9:      end if
10:     min(D) ← minimum(min(D), x)
11:     size(D) ← size(D) + 1
12:     if left(D) = NULL then
13:         s ← 0
14:     else
15:         s ← size(left(D))
16:     end if
17:     if i ≤ s + 1 then
18:         left(D) ← Insert(left(D), i, x)
19:     else
20:         right(D) ← Insert(right(D), i − s − 1, x)
21:     end if
22:
23:     Do the standard colour and height maintenance operation on the R-B
24:     Tree using the modified auxiliary rotate functions given in the following
25:     section
26:
27:     return D
28: end procedure
```

Modified auxiliary rotate function for Insert :

Right-Rotate

Let's assume we have to rotate right about a node y as shown in the figure. While all other procedures of rotation and colour maintainance will be **same** as in an unaugmented R-B tree, the only changes will come in our augmented fields and they are depicted below as follows :



Algorithm 3 Modified Rigt-Rotate

procedure RIGHTROTATE(D,y)
Follow the standard left-rotate procedure with the additional stpes as below:
For the **incr** field -
$val(x) \leftarrow val(x) + incr(x)$
$incr(\beta) \leftarrow incr(\beta) + incr(x)$
$incr(\alpha) \leftarrow incr(\alpha) + incr(x)$
$incr(x) \leftarrow incr(y)$
$incr(y) \leftarrow 0$
For the **min** field -
$min(x) \leftarrow minimum(min(x), min(y))$
$min(y) \leftarrow minimum(min(\beta), min(\gamma), val(y))$
For the **size** field -
$size(y) \leftarrow size(\beta) + size(\gamma) + 1$
$size(x) \leftarrow size(\alpha) + size(y) + 1$
end procedure

Left-Rotate

It is done similarly as Right-Rotate.

Delete(D,i,x)

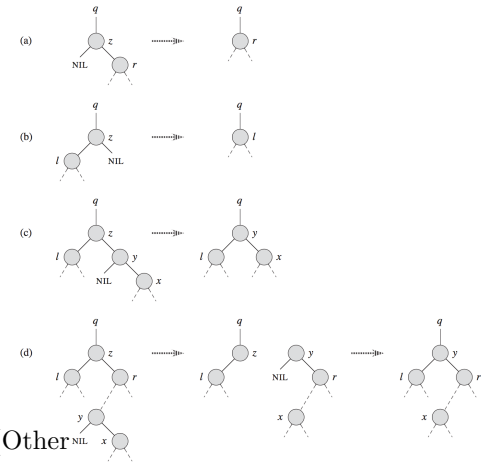
The pseudo-code is as follows:

Algorithm 4 Deleting an element from the **i**-th position

```
1:  procedure DELETE(D,i)
2:      size(D) ← size(D) − 1
3:      if left(D) = NULL then
4:          s ← 0
5:      else
6:          s ← size(left(D))
7:      end if
8:      if i = s + 1 then
9:
10:         Remove(root of the current sub-tree from the orginal tree)
11:
12:     end if
13:     if i < s + 1 then
14:         left(D) ← Delete(left(D), i)
15:     else
16:         right(D) ← Delete(right(D), i − s − 1)
17:     end if
18:
19:     Do the standard colour and height maintenance operation on the R-B
20:     Tree using the modified auxiliary rotate functions given in the following
21:     section
22:
23:     return D
24: end procedure
```

Now, for the **Remove** procedure, we follow the standard removal procedure from a R-B tree with the following changes :

- For removing a node **z** from the tree as shown in the figure, for the **first three** cases, where z is being replaced by its one of its immediate child, let $m = nodereplacingz$ (which is r, l and y respectively in the three cases), $size(m) = size(z)-1$ and $min(m) = minimum(min(m), min(\text{Other child of Z}))$.



• For the 4th case, we need to replace z with its immediate successor (say, y). For doing this, we follow the same procedure as we do in a normal R-B Tree with only a few slight modifications :

- While finding the immediate successor of z, as we travel down the path (from z's right child (r)), we keep on reducing the size of each node from which we travel down to account for the removal of z's immediate successor (y) and its transplant to a higher level which results in its removal from the sub-tree of z's right child.
- We also travel from x (y's right child) back to z's right child (r) while comparing and fixing up the change in minima that might have occurred in the sub-tree of r because of removal of y.

• For We now travel from y (z's replacement) back to the root while comparing and fixing up the change in minima that might have occurred in the tree along this path because of removal of z.

Add(D,i,j, Δ)

The pseudocode and the algorithm for this is **exactly** similar to what we were taught in the class.

Min(i,j)

- We traverse from the i-th element and from the j-th element to their LCA (least common ancestor).
- set $min = val(i)$
- While travelling from i-th element to the LCA, let the current node be **curr** and its parent be **p**. If $p \rightarrow right = curr$, then compare min and p and update min accordingly and then, move further up. But if $p \rightarrow left = curr$, compare $min, pandmin(p \rightarrow right)$ and update min accordingly and then, move further up.
- While travelling from j-th element to the LCA, let the current node be **curr** and its parent be **p**. If $p \rightarrow left = curr$, then compare min and p and update min accordingly and then, move further up. But if $p \rightarrow right = curr$, compare $min, pandmin(p \rightarrow left)$ and update min accordingly and then, move further up.

Time Complexity

Insert

Insert procedure is same as insertion in a R-B tree and the additional processed take O(1) time and hence, Insert takes O(lgn) time, as in a normal R-B tree insertion.

Delete

Deletion procedure is also same as the deletion in a R-B tree and the additional processes either take O(1) time or they involve only one complete or semi-traversal up or down the tree , which we did anyways in a normal R-B deletion and hence, Delete too takes O(lgn) time, as in a normal R-B tree deletion.

Add

Add involves one traversal up the tree and hence, takes $O(lgn)$ time (as discussed in the class).

Min

Similar to Add, Min also involves one traversal up the tree and hence, takes $O(lgn)$ time.

Question 3 (Easy Version)

Designing the Greedy Step

Let **A**=The stated problem with a given height h of the complete binary tree.

Consider the second-lowest level of the tree. For each node v in this level, do the following :

- Find total delay from the root to both the leaf nodes which are children of v, say TD_L and TD_R for the left and right node respectively.
- if $TD_R < TD_L$, increment $D_{Lright}(v)$ (delay in the edge from v to its right child) such that $TD_{Rnew} = TD_L$
- Operate similarly if if $TD_L < TD_R$.
- At the end, contract the node v and make it a leaf node with a new "total $delay$" $= max(TD_L, TD_R)$
- Therefore, total delay enhancement $= |TD_R - TD_L|$ and since delays leading up to the node V are same, therefore, total delay enhancement $= |D_{Lright} - D_{Lleft}|$
- We now get a new complete binary tree with $height = originalheight - 1$, which in fact is **A'**,with a new and updated total delay from the root along each path to the leaf nodes (derived from A, obviously),which is a smaller instance of the problem.
- We continue this procedure till we are left with only one (the root) node.

Establishing a relation between Opt(A) and Opt(A')

Lemma

• There exists an optimal solution for A, say **Opt(A)** such that total delay enhancement on a node $\mathbf{u} = |TD_R - TD_L|$.

This lemma has already been proved in the class for the same problem while implementing a slightly different approach.

0.0.1 Deriving Opt(A) from Opt(A')

To do this, in the tree corresponding to **Opt(A')**, we just expand the node we contracted in the greedy step above.

Now, A' was synchronized and so, all the paths leading up to the leaf nodes from the root nodes had the same total delay. On expanding the leaf nodes so as to gain the tree for A, we will expand each leaf node of A' and to keep A synchronized, we will update the lower delay edge by $|D_{Lright} - D_{Lleft}|$.

This will give us a synchronized circuit which will serve as a solution (not necessarily optimal) for A with

a total delay enhancement= $Opt(A')+\sum_{all\ nodes\ in\ the\ lowest\ level\ of\ A'} |D_{Lright}- D_{Lleft}|$.

Since optimum solution of A means a minimisation of total delay enhancements,thus,

$Opt(A) \leq Opt(A') + \sum_{all\ nodes\ in\ the\ lowest\ level\ of\ A'} |D_{Lright} - D_{Lleft}|$.

0.0.2 Deriving Opt(A') from Opt(A)

Now, using the Lemma, we can say that there will exist an Opt(A) such that in the last level, the enhancement being made by the nodes $= |TD_R - TD_L|$ which $= |D_{Lright} - D_{Lleft}|$ in turn. Thus, when we contract this last level (as mentioned in the greedy step above) in Opt(A), we will get a tree for A' and it will be synchronized since the total delays along both the children of all the nodes in the second last level will be the same (since A is synchronized).

This will give us a synchronized circuit which will serve as a solution (not necessarily optimal) for A' with

a total delay enhancement= $Opt(A)-\sum_{all\ nodes\ in\ the\ lowest\ level\ of\ A'} |D_{Lright}- D_{Lleft}|$.

Since optimum solution of A' means a minimisation of total delay enhancements,thus,

$Opt(A') \leq Opt(A) - \sum_{all\ nodes\ in\ the\ lowest\ level\ of\ A'} |D_{Lright} - D_{Lleft}|$.

• **Conclusion:** Combining the results above, we can safely say that

$Opt(A) = Opt(A') + \sum_{all\ nodes\ in\ the\ lowest\ level\ of\ A'} |D_{Lright} - D_{Lleft}|$.

This establishes our Greedy method.

Algorithm

The algortihm is pretty straight-forward and is evident from the greedy step described above. It is as follows:

- For every node **v** in the second last level of the tree (in case there is only one level, it means there is only the root and our problem is solved),we find total delay from the root to both the leaf nodes which are children of v, say TD_L and TD_R for the left and right child node respectively.
- if $TD_R < TD_L$, increment $D_{Lright}(v)$ (delay in the edge from v to its right child) such that $TD_{Rnew} = TD_L$
- Operate similarly if if $TD_L < TD_R$.
- At the end, contract the node v and make it a leaf node with a new "total $delay$" $= max(TD_L, TD_R)$
- Increment the total delay enhancement by $|TD_R - TD_L|$ and since delays leading up to the node V are same, therefore, this increment is also equal to $|D_{Lright} - D_{Lleft}|$.
- We now get a new complete binary tree with $height = originalheight - 1$ and updated total delay from the root along each path to the leaf nodes which is a smaller instance of the problem.
- We continue this procedure till we are left with only one (the root) node which serves as our base case and the problem is solved.
- We report the "total delay enhancement" in the end as our answer.