Data Structures and Algorithms

Varun Khare (150793) and Anmol Porwal (150108)

9 March 2017

1 Infinite Array Problem

Let A be the given infinite array and s be the value to be found and n be the total number of integer values in the array. Consider the rest of entries to be NULL.

1.1 Pseudo Code

Algorithm 1 Find value s in infinite array

```
1: procedure FIND(A,s)
2:     if A[0] == s then
3:         Return TRUE
4:     end if
5:     2_pow_high ← 1;
6:     while A[2_pow_high − 1] < s || A[2_pow_high − 1] == NULL do
7:         2_pow_high = 2 × 2_pow_high
8:         if A[2_pow_high − 1] == s then
9:             ReturnTRUE
10:        end if
11:    end while
12:    2_pow_low = 2_pow_high / 2
13:    Return Binsearch (A,2_pow_low,2_pow_high)
14: end procedure
```

Algorithm 2 To perform binary search between 2 indices in an array

```
1: procedure BINSEARCH(A,low,high)
2:     mid = (low + high)/2
3:     while low < high do
4:         if mid == NULL || mid > s then
5:             high = mid -1
6:         else if mid == s then
7:             Return TRUE
8:         elselow = mid + 1
9:         end if
10:    end while
11:    if low == high  then
12:        if A[low]==s then
13:            Return TRUE
14:        end if
15:    else if A[low+1] == s then
16:        Return TRUE
17:    end if
18:    Return FLASE
19: end procedure
```

1.2 Proof of correctness

Consider any element A[i] in the sorted array A with null values in the last. For the sake of proof we can treat NULL elements of the array as ∞. They are present in the last of the array so are biggest of all finite elements of the array. Thus, it is safe to assume NULL as ∞ for proof.

1.2.1 "s" is present in the output array of Find:

Utilising sorted Array we know if $A[i] < s \Rightarrow\ A[j] < s\ \forall j \in \{0, 1, 2, ...i\}$ and $A[i] > s \Rightarrow\ A[j] < s\ \forall j > i\ and j \in N$ Thus until $s > A[i]$ we take i as higher powers of 2 and take this value of i which is a power of 2 as upper index of the array to be fed in Binsearch and since $s > A[i/2]$ as i is the smallest power of 2 such that $s < A[i]$. so i/2 is the lower index of array to be fed in Binsearch. So s will be present between these two indices in the array.

1.2.2 Binary search returns correct output:

Binary search always checks for the required value and will definitely return it if it exists in the sorted array.
Now, if $A[i] < s \Rightarrow\ A[j] < s\ \forall j \in \{0, 1, 2, ...i\}$ due to sorted array
This proves that the value s (if it exists) in the array can only belong to {i+1,i+2,...end}. Similarly for $A[i] > s \Rightarrow\ A[j] > s\ \forall j \in \{i + 1, i + 2, ...end\}$
Since these are the reduced arrays for next iteration s is contained in the reduced array whenever $A[i] \neq s$ This justifies reduction of array to be searched to one of two halves after each iteration.
Since the value s is always contained in the reduced array that eventually will reduce to 1 or zero, the final reduced Array after loop terminates must contain s. The final Array of the output will be Empty if s does not lie in array. So Binary search will always return TRUE if s is present or else return FALSE.

1.3 Time Complexity Analysis

- Binsearch(A,low,high)
  - Line 2,11-18 : O(1)
  - Line 3-10 : In each iteration of the loop (high-low) decreases by half hence the loop iterates O(log(high-low)) times. And lines 4-9 have O(1) complexity hence O(log(high-low)) time complexity.

  Hence Time Complexity of Binsearch = O(log(high-low))

- FIND(A,s)
  - Line 2-5,12 : O(1)
  - Line 6-11 : The while loop runs for at most $\lceil log(n) \rceil$ number of times.And each iteration of the loop takes O(1) time hence O($\lceil log(n) \rceil$) time.
  - Line 13: now since (2_pow_high - 2_pow_low) = 2_pow_high / 2 = $\lceil log(n) \rceil$ / 2. Therefore time complexity = O(log($\lceil log(n) \rceil$)).

  Hence time complexity of Find = O($\lceil log(n) \rceil$) + O(log($\lceil log(n) \rceil$)) = O($\lceil log(n) \rceil$).

2 Maximum Possible Area of Banner Question

2.1 Pseudo-Code

Let n be the total number of buildings (0,1,2,...n-1) under the ownership of proprietor. H[n] and B[n] represent the heights and widths of the buildings respectively. Therefore, for i^{th} building we have:

H[i]=height of the building
B[i]=width of the building

Algorithm 3 Find maximum possible area for banner

```
1: procedure MaxArea(H[ ],B[ ],n)
2:     Ar ← 0; MaxAr ← 0; h ← 0; i ← 0; Stack S ← Null; Stack L ← Null;
3:     Left[n] ← {0}, Right[n] ← {0};
4:     while i < n do
5:         l ← 0;
6:         if !isEmpty(S) && H[i] ≤ Top[S] then
7:             while ( !isEmpty(S) && (H[i] ≤ H[Top(S)]) ) do
8:                 Pop(S);
9:                 l ← l + Pop(L);
10:            end while
11:            Push(S, i);
12:            h ← H[i]; l ← l + B[i];
13:            Push(L, l);
14:            Left[i] ← l × h;
15:        else
16:            Left[i] ← H[i] × B[i];
17:            Push(S, i);
18:            l ← B[i];
19:            Push(L, l);
20:        end if
21:        i = i + 1;
22:    end while
23:    i ← n − 1;
24:    empty Stack S;
25:    empty Stack L;
26:    while i ≥ 0 do
27:        l ← 0;
28:        if !isEmpty(S) && H[i] ≤ Top[S] then
29:            while ( !isEmpty(S) && (H[i] ≤ H[Top(S)]) ) do
30:                Pop(S);
31:                l ← l + Pop(L);
32:            end while
33:            Push(S, i);
34:            h ← H[i]; l ← l + B[i];
35:            Push(L, l);
36:            Right[i] ← l × h;
37:        else
38:            Right[i] ← H[i] × B[i];
39:            Push(S, i);
40:            l ← B[i];
41:            Push(L, l);
42:        end if
43:        i = i − 1;
44:    end while
45:    i ← 0;
46:    MaxAr ← 0;
47:    while i < n do
48:        TempAr ← Left[i] + Right[i] − H[i] × B[i];
49:        if MaxAr < TempAr then
50:            MaxAr ← TempAr;
51:        end if
52:        i = i + 1;
53:    end while
54:    Return MaxAr;
55: end procedure
```

Algorithm 4 Stack Functions

```
1: procedure PUSH(Stack S,Value val)
2:     create node ν;
3:     ν.value ← val;
4:     if isEmpty(S) then
5:         S ← ν;
6:     else
7:         ν.next ← S;
8:         S ← ν;
9:     end if
10: end procedure
11: procedure POP(Stack S)
12:     if !isEmpty(S) then
13:         v ← S.next;
14:         S ← S.next;
15:         freev;
16:     end if
17: end procedure
18: procedure TOP(Stack S)
19:     return S.value;
20: end procedure
```

2.2 Proof Of Correctness

We give the proof by induction for the above algorithm and hence justify that algorithm 2 indeed returns the maximum possible area of the banner. Let n be the number of the buildings under the ownership of proprietor.

Induction hypothesis

P(n): Left[n] stores the maximum area of banner that ends at building with index "n" encompassing the building entirely.

Proof:
P(1): After first iteration we have traversed only a single building so the largest possible banner covers the entire building.
$\Rightarrow$ area of banner = $H[0] \times B[0]$
And Left[0] is indeed $H[0] \times B[0]$. Thus, P(1) is true.
Let **P(j)** be true $\forall j \in \{0, 1, 2...k\}$. To justify the algorithm we need to prove P(k+1) is true.

P(k+1): $Left[j]$ stores the maximum possible banner ending at building $j \forall j \in \{0, 1, ...k\}$. Knowing this there are two cases for the building $k+1$ when $H[k+1] \geq H[k]$ and when $H[k+1] < H[k]$:

- **Case 1:** $H[k+1] > H[k]$
  In this case the banner can only have an area across building k+1 only.
  As any extension of this banner to left will bring the rectangular part outside of complex C_{k+1}.

- **Case 2:** $H[k+1] \leq H[k]$
  In this case an entirely new banner has to be created for C_{k+1} with height = H[k+1] less than or equal to banner for C_k. So the maximum possible left banner in this manner will be the one encompassing all the buildings of height greater than or equal to H[k+1].
  We used stack of buildings to extract this property. Stack S stores the heights (via index) of the banners with heights less than the pushed values.
  Stack L stores the corresponding length of each of those banners. This essentially means L stores the lengths of all the banners of increasing heights generated while traversing buildings till then. This is true as only the values of the generated banners for which Left[i] is evaluated are pushed to the stacks. Since P(i) ($i < k+1$) is true, left[i] denotes the Maximum possible area on the left side enclosing building i. Note that each entry of L represents there is no banner (and hence building) of greater height immediately before it (otherwise we could include that for creating the maximum size banner for building i and then left[i] won't be largest area which is contradiction). this proves heights of banners in the stack are in increasing order
  $\Rightarrow$ All the buildings inside the banners of the stack S have $height \geq banner\ height$.
  So if the current building has height less than the Top banner height, its height will be less than all the buildings encompassed by that banner. This means the maximum left banner encompassing building k+1 would include this banner's width completely. Hence we pop off that banner from stack and add its width to the banner of current building.We would continue Popping the elements of the stack L and S until we find a building s.t. $H[i] < H[k+1]$.
  When the banner comes whose height is less than the height of building k+1 then the banner can't extend further (as in case 1) and we would stop. The increasing order of heights in the stacks confirm that there is no building of height lesser than the current building before we stop popping. Thus we get the immediate previous smaller height building which determine the width of the banner as discussed above. We push the new Banner to stack for further evaluation and have the correct defined value of the area in left[k+1].

Hence P(k+1) is true.**Thus P(n) is true.**

Same proof goes for Right[i] which stores the maximum possible area from i building towards right encompassing it entirely. Actually its same we just traverse backwards.
So the total Area (TempAr) of the encompassing banner, for building i, from both the sides would be union of left[i] and

right[i] banners.
We know that such rectangle would cover at least one of the buildings of complex completely. Otherwise, we can increase the size of banner by giving it the height of the minimum height building which would prove it is not the maximum area rectangle. At the same time the rectangle would be maximum encompassing area for the smallest height building. Thus the answer to the problem would be amongst the values of the TempAr evaluated above. Thus, maximum(TempAr) is the correct answer.MaxAr updates its value iff TempAr is greater than MaxAr at any instant. So after loop execution MaxAr has the largest possible banner for the complex C_n ie the entire complex. Hence the algorithm is justified.

2.3 Time Complexity Analysis

2.3.1 Stack Functions analysis

For Algorithm 2:

Push(S,val)	lines 2-9 take O(1) each $\Rightarrow O(Push(S,val)) = O(1) \times 6 = O(1)$
Pop(S)	lines 12-16 take O(1) each $\Rightarrow O(Pop(S) = O(1) \times 4 = O(1)$
Top(S)	line 19 take O(1) $\Rightarrow O(Top(S)) = O(1)$

2.3.2 Algorithm 1 Analysis

Lines 2-3 take O(1) each. Lines 5-21, 27-43 and 48-52 run n times. Here we have two nested while loops (26-44 and 47-53).
Considering first one of these loops
For each iteration of outer while loop we have O(1) for lines 5-6 and lines 11-21. Also let inner while loop (lines 7-10) run ν_i times {with each iteration taking O(1) time } for i^{th} iteration of outer while loop. Thus, time complexity(T), for i iteration is

$$T = O(1) \times 13 + O(\nu_i) = O(\nu_i) + O(1) \tag{1}$$

Total time complexity of This loop (Time)

$$\begin{aligned} Time &= \sum_{i=0}^{n-1} (O(\nu_i) + O(1)) \\ &= \Big\{ \sum_{i=0}^{n-1} O(\nu_i) \Big\} + O(n) \\ &= O\Big(\sum_{i=0}^{n-1} \nu_i \Big) + O(n) \end{aligned}$$

Since each ν_i represents the number of non-null pops in the stack S, $\sum_{i=0}^{n-1} \nu_i$ must be less than or equal to number of $Push_s$ to the stack. Now Push(S,i) occurs only once for each iteration of outer while loop which gives total insertions to the stack as n

$$\sum_{i=0}^{n-1} \nu_i \leq \ number\ of\ Push_s\ to\ S = n$$

$$\sum_{i=0}^{n-1} \nu_i \leq n \Rightarrow O\Big(\sum_{i=0}^{n-1} \nu_i \Big) = O(n)$$

Hence overall time complexity(Time) of this loop is O(n)+O(n) = O(2n) = O(n).
Similarly following for the other loop (26-48), we have O(n). Loop (47-53) takes O(n) as each line of that loop takes O(1).
Thus overall Time complexity of the algorithm is **O(n)+O(n)+O(n)+O(1)=O(n)**

3 Augmented Red Black Trees

Here we have added 3 new fields in each node of the Red-Black-Trees:

- Pointer to predecessor : Node * pred (The node with smallest value has pointer to NULL)
  This is used for finding the predecessor of a node in O(1) time.
- Pointer to parent node : Node * parent (Root node has pointer to NULL)

- Whether the node is right or left child of it's parent : bool isLeft(true when node is the left child and for root node we take it to be NULL by convention)
  The above 2 attributes are used to find predecessor of a node without using the "pred" attribute.

now input is root node "root", the value whose predecessors are to be found "s", array "A" to store the predecessors and "k".
Here if there are less than k predecessors of the given value let us say k_1 then our algorithm stores these k_1 values in the array "A".

3.1 Pseudo Code

Algorithm 5 function to store k predecessors in an array

```
1: procedure KPredecessors(root,s,k,A )
2:     while root ≠ NULL do
3:         if root.val == s then
4:             Break
5:         else if root.val < s then
6:             root ← root.right
7:         else
8:             root ← root.left
9:         end if
10:    end while
11:    i ← 0
12:    while i < k && root ≠ NULL do
13:        A[i] ← root.pred.val
14:        root ← root.pred
15:        i++
16:    end while
17: end procedure
```

Algorithm 6 log(n) time algorithm to find predecessor(similar algorithm for successor)

```
1: procedure Predecessor(v)
2:     if v.left ≠ NULL then
3:         Return Max(v.Left) \\Max(v.Left) returns the maximum value in the left subtree of v.
4:     else
5:         while v.isLeft do
6:             v ← v.parent
7:         end while
8:         Return v.parent
9:     end if
10: end procedure
```

3.2 Proof of correctness

We give the proof of correctness of the algorithm by showing that the $(i-1)^{th}$ index of the array indeed stores the i^{th} predecessor of the given element. For this we just need to show:
1.) That the "pred" field of a node stores the predecessor of the node i.e. the log(n) time algorithm gives the correct predecessor.
the predecessor is the maximum element (bottom right element) in the left subtree of the node if it is not empty. If it is empty we go up until we find a node which is the right child of it's parent(coz if it's the left child then it is less than the parent) then the parent of this node will be the required predecessor.
2.) We find the correct node which stores the given value.
for this we just need to prove that in every iteration of 1st while loop the value to be found is in the subtree rooted at "root" so initially it would either be equal to value of root or in the right subtree or left subtree. if it is equal to value in root we are done if value in root<s then by property of BST s is in right subtree of root and hence b updating "root" to right subtree would ensure that the value is indeed in the subtree rooted at "root" else if s > value of root then we update "root" as left subtree of "root" which would again ensure that subtree rooted at "root" node contains the value s.
Hence our algorithm correctly stores predecessors of value s.

Let us see the modifications and time complexity of the following operations in the new augmented red-black-trees:

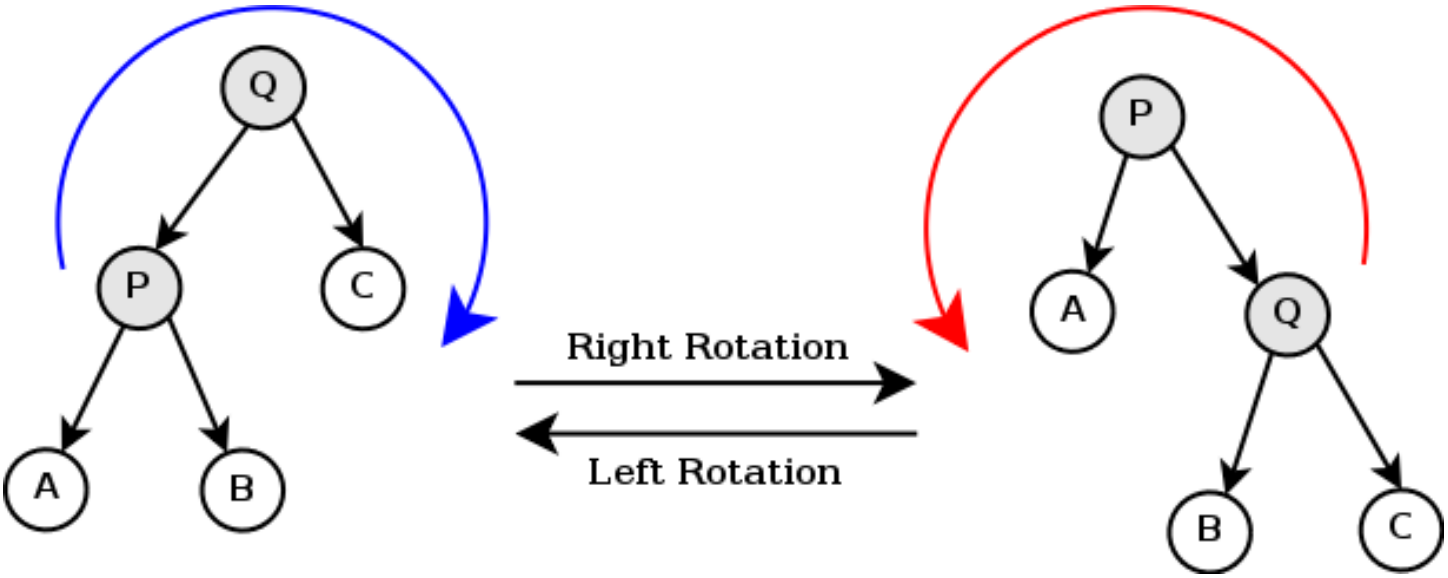

Figure 1: Tree Rotations

- Rotations:
  - Pointer to predecessor : It need not be updated while performing rotations as the predecessor of a particular node remains same.
  - Pointer to parent node : It needs to be updated for three nodes on a single rotation for example in the Figure 1 on any of the rotations we just need to modify the parents of Q,P and B. Hence it takes just O(1) time.
  - Whether the node is right or left child of it's parent : similarly it needs to be updated for at most 3 nodes P,Q and B only hence it again takes o(1) time only.

  Hence we can see that the time complexity of rotations remain the same being O(1).

  In Add and Delete during balancing we perform rotations or colour changes now we have seen how to handle rotations, and colour changes need no changes in the new modifications hence for add and delete we just need to modify the 1st steps of simple insertion and deletion.

- Add: Insert the new node with red colour just as we did previously.
  - Pointer to predecessor(Let v_{i-1}) : we need to modify this field for 2 nodes the inserted node(Let v_i) and it's successor(Let v_{i+1}) . For the inserted node find it's successor using the O(log(n)) tree traversal algorithm described in class. Now store v_{i+1}.pred in v_i.pred and change v_{i+1}.pred to v_i.
  - Pointer to parent node :
  - Whether the node is right or left child of it's parent :
    The above 2 attributes are only needed to be modified for the inserted node which can easily be modified by making little modification in the algorithm discussed in class.

- Delete:
  - Pointer to predecessor :We need to modify just the "pred" field of the successor of the deleted node. we do so by finding the successor of the node to be deleted using the O(log(n)) tree traversal algorithm discussed in class and then updating it's "pred" field to the "pred" value of the node to be deleted.
  - Pointer to parent node : NA
  - Whether the node is right or left child of it's parent : NA

  Hence for both add and delete we just need an extra O(log(n)) time to maintain these new attributes hence the time complexity of add and delete remains same.

- Query: No update needs to be performed to this operation hence it's time complexity also remains the same.

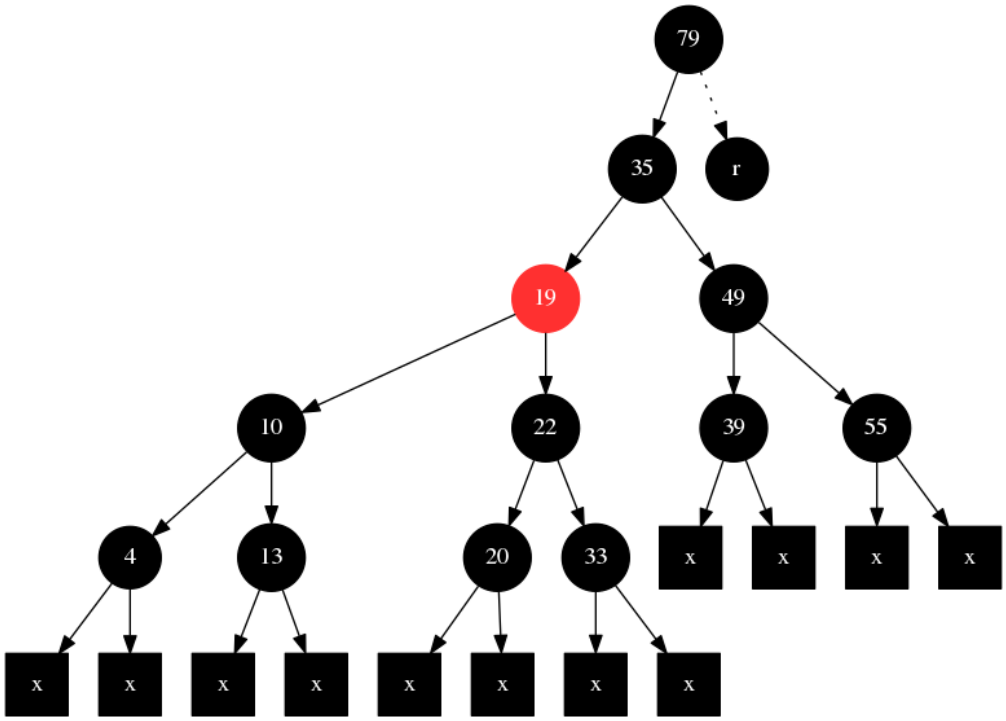

Figure 2: The Given Tree

3.3 time Complexity Analysis

- Line 2-10 : the while loop iterates at most (height+1) times and since it is a red black tree it becomes O(log(n)) (n being total number of nodes) lines 3-9 are O(1) therefore overall it becomes O(log(n))
- Line 11 : O(1)
- Line 12-16 : the while loop iterates at most k number of times and lines 13-15 are O(1) hence overall it becomes O(k)

overall time complexity of KPredecesors = O(log(n)) + O(k) = O(log(n)+k)

4 Red Black Trees

4.1 Q1

The subsequent stages of deleting 55 is shown along the figures 2 ->3 ->4 ->5 ->6

4.2 Q2

Figure 7 represents the tree after insertion of node 34 which is done just by inserting 34 with red colour.

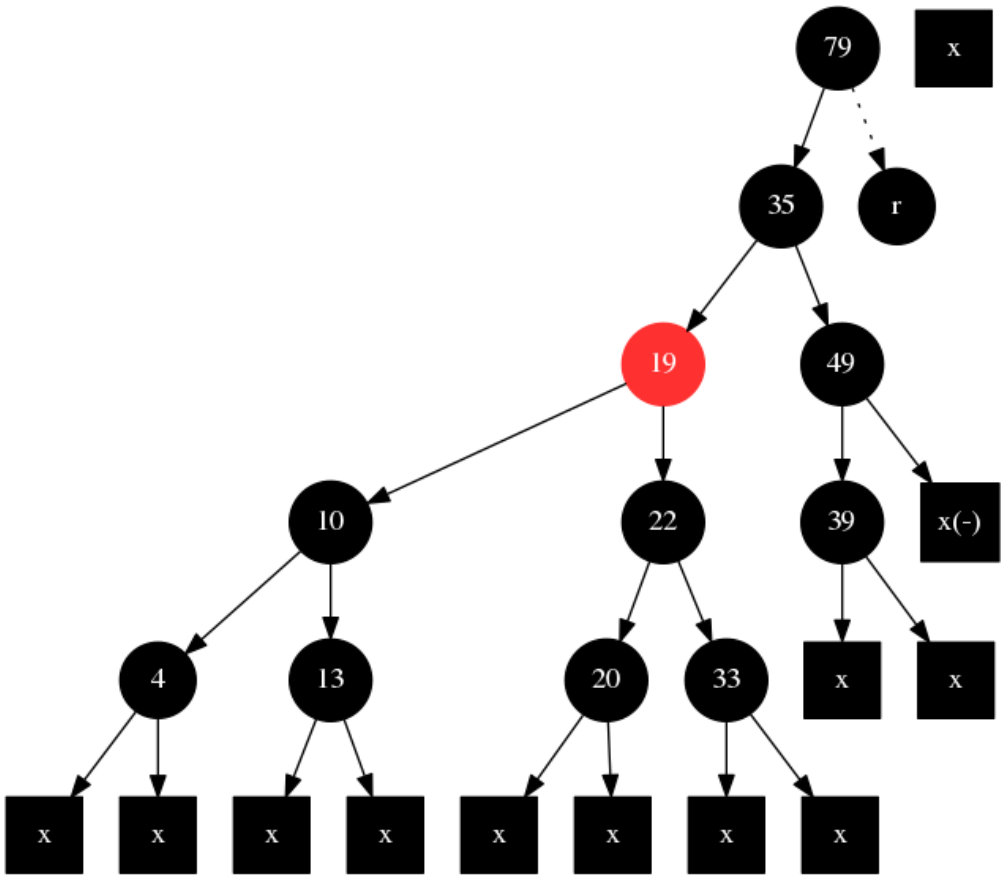

Figure 3: Tree after deleting 55

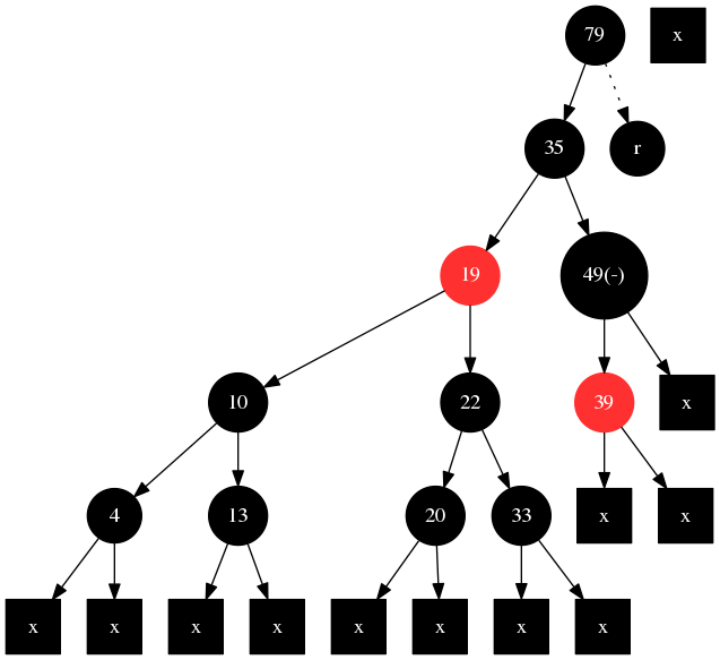

Figure 4: making node 39 red to make subtree rooted at 49 to be unbalanced(by colour)

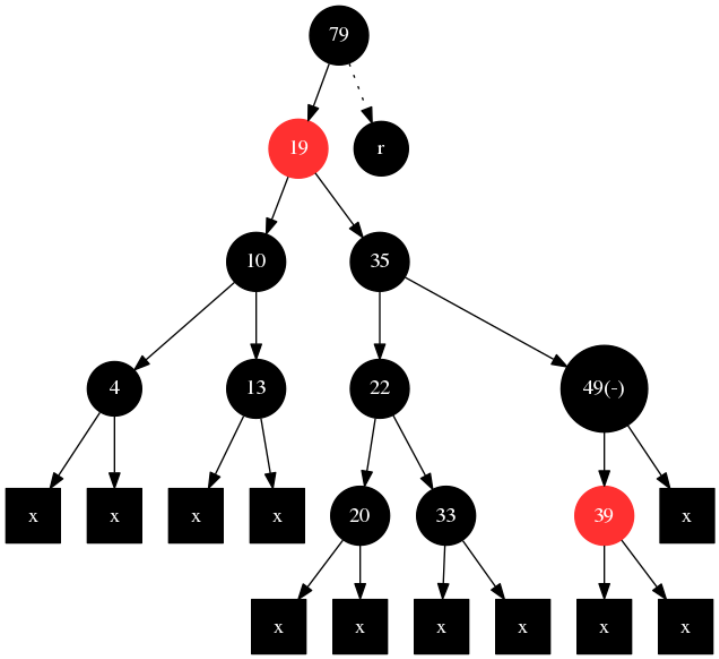

Figure 5: Right rotation about node 35 this makes subtree rooted at 10 to become unbalanced(by colour) along with one at 49

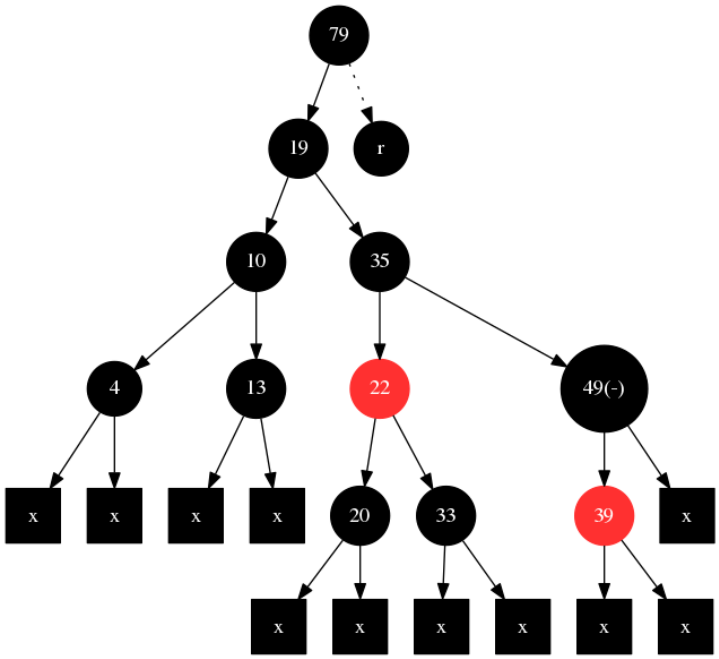

Figure 6: Swapping colours of 19 and 22 to make the 2 subtrees balanced

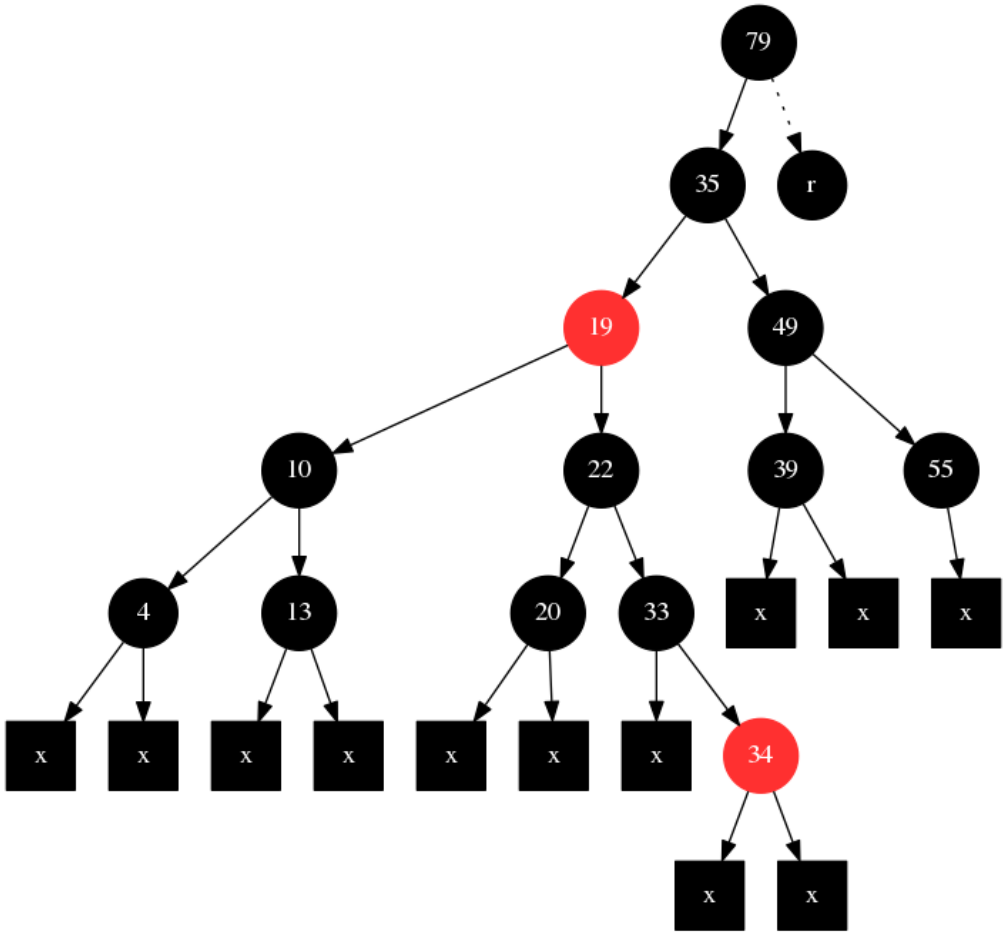

Figure 7: Caption