CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Nov

Scribe: Soham Chatterjee

soham.chatterjee@tifr.res.in
Website: sohamch08.github.io

Contents

Chapter 1

Red Black Tree

A red-black tree is a special type of binary search tree with one extra bit of storage per node, its color which can be either red or black. Also, we keep the tree approximately balanced by enforcing some properties on the tree.

Definition 1.1: Perfect Binary Tree

It is a Binary Tree in which every internal node has exactly two children and all leaves are at the same level.

Lemma 1.1

Every perfect binary tree with k leaves has $2k - 1$ nodes (i.e. $k - 1$ internal nodes).

Definition 1.2: Red Black Tree

A red-black tree is a binary tree with the following properties:

- Every internal node is key/NIL node. Every leaf is a "NIL" node.
- Each node (NIL and key) is colored either red or black.
- Root and NIL nodes are always black.
- Any child of a red node is black.
- The path from root to any leaf has the same number of black nodes.

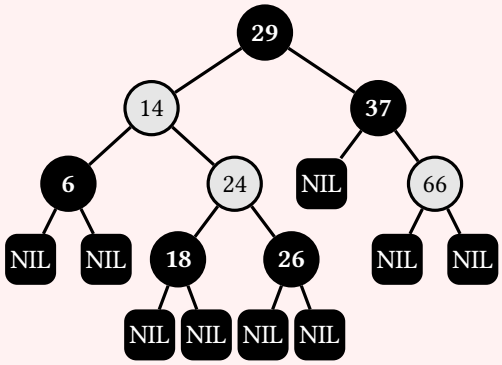


Figure 1.1: A Red Black Tree

We call the number of black nodes on any simple path from but not including a node x down to a leaf the *black-height* of the node, denoted by $bh(x)$. We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values.

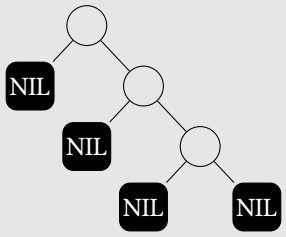
Lemma 1.2

A Red-Black Tree with n internal nodes or key nodes has height at most $O(\log n)$.

Proof: We will first show that for any subtree rooted at node x contains at least $2^{bh(x)} - 1$ internal nodes. We will show this using induction on the height of the tree. For the base case let height of x is 0. Then x must be a leaf. Therefore, the subtree rooted at x has at least $bh(x) = 0$. Hence, $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nodes which is true. For inductive step let x has some positive height, and it is an internal node of the R-B Tree. Now x has two children. Hence, each child has black-height either $bh(x)$ or $bh(x) - 1$. By inductive hypothesis, the subtrees rooted at the children of x have at least $2^{bh(x)-1} - 1$ internal nodes. Thus, subtree rooted at x has at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ internal nodes.

Now if the R-B tree has height h. Then any path from the root to a leaf at least half the nodes including the root must be black. So $bh(root) \geq \frac{h}{2}$. Thus, $n \geq 2^{\frac{h}{2}} - 1 \implies h \leq 2\log(n+1)$. Hence, we have the lemma. ■

Note:-



Not all trees can be colored in a way that satisfies the properties of a red-black tree. Consider the following tree:

In this example the root has to be black. The other two internal nodes can not be black since otherwise the path from the leaf of the root to root has only 2 black nodes but in the path from bottom most leaf to root will have 3. Then those two internal nodes has to be red. But that violates the property that a red node can not have a red child. Hence, this tree can not be colored in a way that satisfies the properties of a red-black tree.

Since by the lemma the R-B tree has height at most $O(\log n)$ and it is a binary search tree we can perform search of a node using Find in $O(\log n)$ time. So now we will focus on the insertion and deletion operations in a red-black tree. To insert or delete a node in a red-black tree we will rotations to balance the tree again. So first we will visit rotations.

1.1 Rotation

A rotation is a local operation that changes the structure of a binary tree without violating the binary search tree property. There are two types of rotations: left rotation and right rotation.

When we do a left rotation on a node we assume that its right child is not NIL. The left rotation "pivots" around the link from the node to its right child and makes the right child the new root of the subtree with the ode as its left child. Similarly, we can explain the right rotation. The rotations behave like the following:

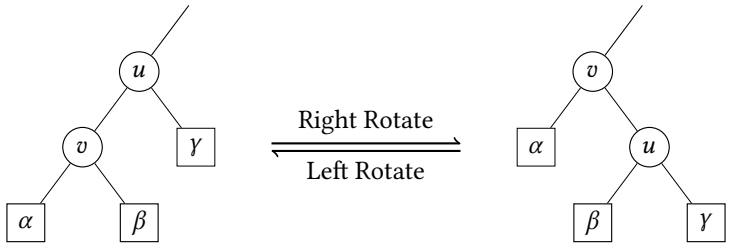


Figure 1.2: Left and Right rotate about $u - v$

Algorithm 1: Left-Rotate(T, x)

```
1  y ⟵ x.right
2  x.right ⟵ y.left
3  if y.left ≠ NIL then
4  |  y.left.parent ⟵ x
5  y.parent ⟵ x.parent
6  if x.parent == NIL then
7  |  T.root ⟵ y
8  else if x == x.parent.left then
9  |  x.parent.left ⟵ y
10 else
11 |  x.parent.right ⟵ y
12 y.left ⟵ x
13 x.parent ⟵ y
```

Algorithm 2: Right-Rotate(T, x)

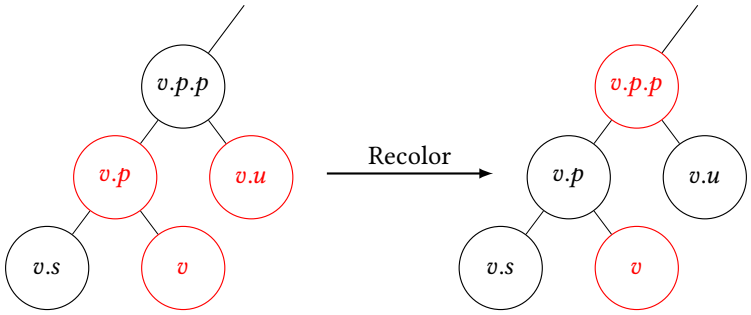
```
1  y ⟵ x.left
2  x.left ⟵ y.right
3  if y.right ≠ NIL then
4  |  y.right.parent ⟵ x
5  y.parent ⟵ x.parent
6  if x.parent == NIL then
7  |  T.root ⟵ y
8  else if x == x.parent.left then
9  |  x.parent.left ⟵ y
10 else
11 |  x.parent.right ⟵ y
12 y.right ⟵ x
13 x.parent ⟵ y
```

Both Left-Rotate and Right-Rotate take $O(1)$ time. Only some constantly many pointers are changed by rotation all other attributes in a node remain the same.

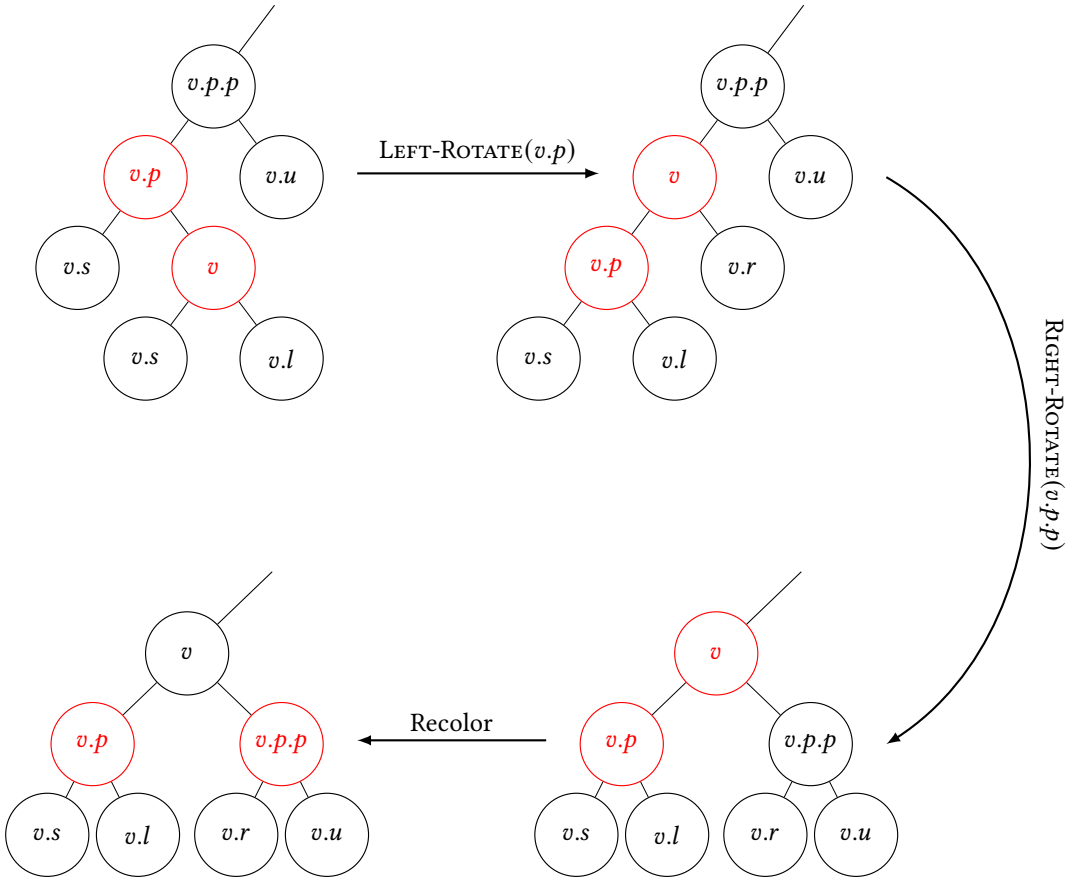
1.2 Insertion

We will now describe how to insert a node in a red-black tree in $O(\log n)$ time. We will insert the node in the tree in place of a leaf replacing a NIL node. After that we will color the node red. Let the node added is v. We define the attribute *uncle* which is basically sibling of the parent. Now two cases can happen:

Case I: $v.uncle.color$ = Red: Then $v.parent.parent$ is black. In this case we can recolor $v.parent.parent$ to red and both $v.parent$ and $v.uncle$ to be black. This will preserve the number of black nodes in any simple path from root to any leaf. Now the color of $v.parent.parent$ is red, and therefore we iterate the same process on $v.parent.parent$.



Case II: $v.uncle.color$ = Black: In this case we need two rotations. First we do a left rotation on $v.parent$. After that we do a right rotation on $v.parent.parent$. After the rotations, we recolor the nodes. The color of $v.parent.parent$



will be black and the color of $v.parent$ will be red. The color of v will remain red. This will preserve the number of black nodes in any simple path from root to any leaf. And this case now stabilizes the tree and we can stop the process.

So analyzing the insertion process we can insert a node in a red-black tree and using the two cases we can recolor the nodes the balance the tree.

Algorithm 3: RB-INSERT(T, v)

```
1  y ⟵ NIL, x ⟵ T.root
2  while x ≠ NIL do
3      y ⟵ x
4      if v.key < x.key then
5          x ⟵ x.left
6      else
7          x ⟵ x.right
8  v.parent ⟵ y
9  if y == NIL then
10     T.root ⟵ v
11 else if v.key < y.key then
12     y.left ⟵ v
13 else
14     y.right ⟵ v
15 v.left ⟵ NIL, v.right ⟵ NIL, v.color ⟵ RED
16 RB-INSERT-FIXUP(T, v)
```

Algorithm 4: RB-INSERT-FIXUP(T, v)

```
1  while v.parent.color == RED do
2      if v.parent.parent == NIL then
3          v.parent.color ⟵ BLACK
4          Break
5      vu ⟵ v.parent.parent.right          // Uncle
6      vpp ⟵ v.parent.parent
7      if vu.color == RED then
8          v.parent.color ⟵ BLACK          // Case I
9          vu.color ⟵ BLACK
10         vpp.color ⟵ RED
11         v ⟵ vpp
12     else
13         LEFT-ROTATE(T, v.parent)        // Case II
14         RIGHT-ROTATE(T, vpp)
15         v.color ⟵ BLACK
16         vpp.color ⟵ RED
17         Break
```

Since the case I can happen at most $O(\log n)$ times as each use of case I increase the height by 2 the while loop can run at most $O(\log n)$ times. Therefore, insertion of a node in a red-black tree takes $O(\log n)$ time.

1.3 Deletion

Like insertion, deletion of a node involves recoloring and rotations to maintain the properties of a red-black tree. Here we will use a notion of double-black color. In the deletion we will use something called in-order traversal of the binary tree and use successor and predecessor of a node in the traversal.

Definition 1.3.1: In-Order Traversal

In-Order Traversal of a binary tree is a traversal where:

- Recursively traverse the current node left subtree.
- Visit the current node.
- Recursively traverse the current node right subtree.

The in-order *successor* (*predecessor*) of a node is the next (previous) node in the in-order traversal of the tree.

Observation 1.1. *For any node x the in-order successor of x is the leftmost node in the right subtree of x. Similarly, the in-order predecessor of x is the rightmost node in the left subtree of x.*

To delete a node x we will replace its *key* by the key of its in-order successor or predecessor (say y) and then delete y i.e. after replacing the key of x by the key of y, it will still have the color of $x.color$. We replace with in-order successor unless x has no right child. In that case we replace with in-order predecessor. If x has no children then we have $y = x$.

Note:-

y is either a non-NIL leaf or has exactly one child.

1. y has a child then child must be colored red since otherwise the NIL child of y and any NIL node in the subtree rooted at child of y will have different black-height. Therefore, y must be colored black. Hence, we replace y by its child and color it black.

2. y is a non-NIL leaf and its colored red. Then we can simply remove y from the tree.

So the only case remained to analyze is when y is a non-NIL leaf and colored black. Now the situation is complicated since removing y would create black-height imbalance in the tree.

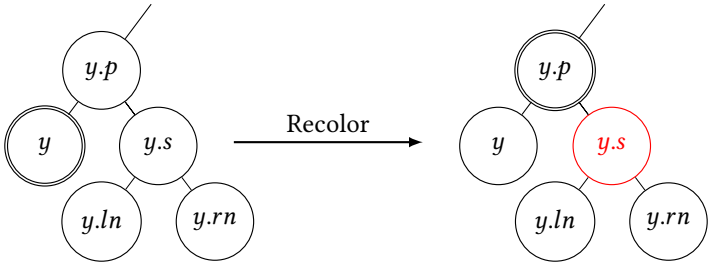
Observation 1.2. *If* y.color *is black then* y *must have a sibling since otherwise sibling of* y *is NIL. Then that NIL node and any NIL node in the subtree rooted at* y *will have different black-height.*

So we replace y with a NIL node and color it *double-black* which will be counted has 2 black nodes to maintain the black-height. Now we will resolve the double-black color by rotation, recoloring or pushing up the double-black color. Now we will use the following pointers

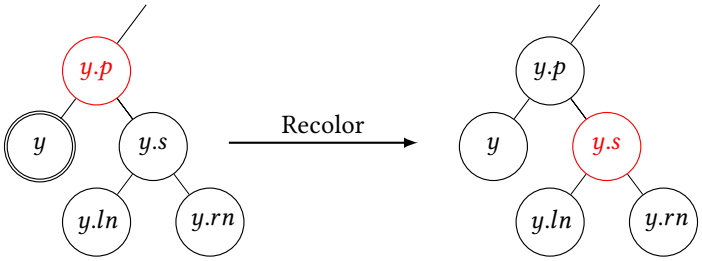
- $y.sibling$ to denote the sibling of y.
- $y.left\text{-}nephew$ and $y.right\text{-}nephew$ to denote the left and right child of $y.sibling$.

We will use the following cases to resolve the double-black color:

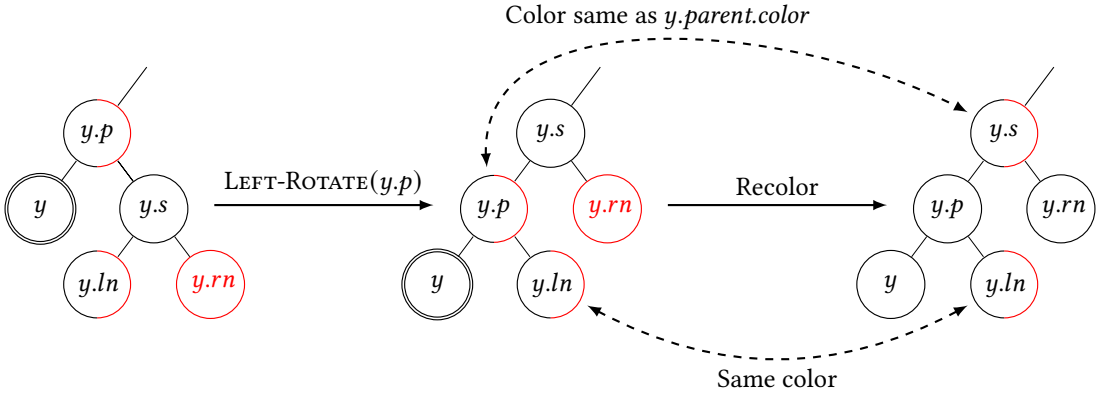
Case I: $y.sibling, y.parent, y.left\text{-}nephew, y.right\text{-}nephew$ are Black. In this case we can move the double black node to the parent of y and recolor the y has black node and sibling of y red color.



Case II: $y.sibling, y.left\text{-}nephew, y.right\text{-}nephew$ are Black & $y.parent$ is Red. Here we recolor $y.parent$ to black and $y.sibling$ to red. This will preserve the number of black nodes in any path from root to any leaf. So we stop.

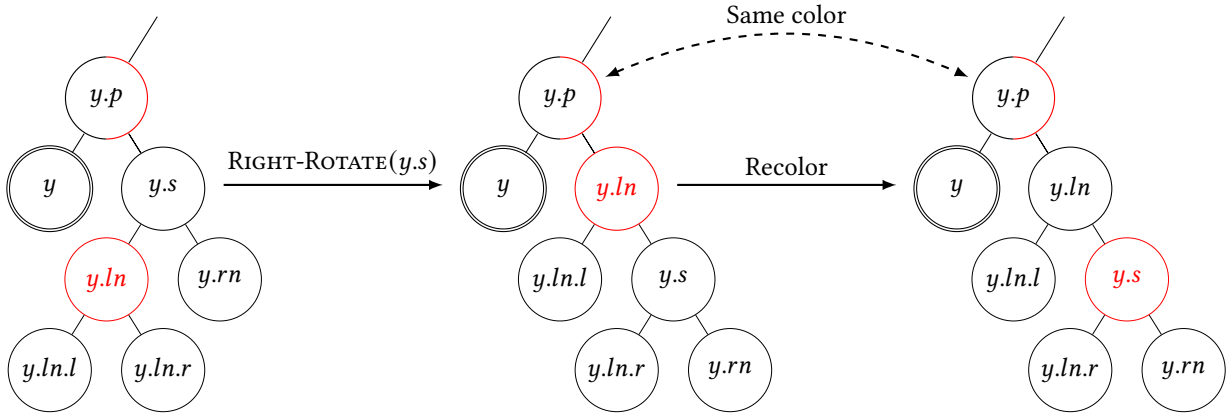


Case III: $y.sibling$ is Black and $y.right\text{-}nephew$ is Red. Then we do a LEFT-ROTATE on $y.parent$. And we recolor $y.parent$



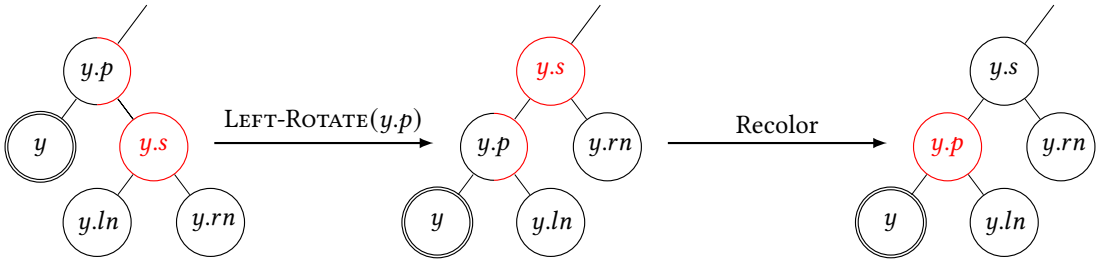
to black and $y.right\text{-}nephew$ to red and y to black. We color $y.sibling$ to the same color as $y.parent$. And now we stop.

Case IV: *y.sibling*, *y.right-nephew* are Black & *y.left-nephew* is Red. Therefore, both the children of *y.left-nephew* have color black. Here we first do a RIGHT-ROTATE on *y.sibling*. Then we recolor *y.left-nephew* to black and *y.sibling*



to red. Now we have exactly the same situation as in Case III. So we follow the steps of Case III.

Case V: *y.sibling* = Red. In this case *y.left-newphew* and *y.right-nephew* must be black. Then we do a LEFT-ROTATE on *y.parent*. Then we recolor *y.parent* to black and *y.sibling* to red. Now we have the sibling of *y* has color black.



So we are now in one of the previous cases. So we can follow the suitable case to resolve.

This completes the description of the deletion process in a red-black tree. Now notice every time we are pushing the double-black color up the tree, or we are stopping. Hence, it only takes $O(\log n)$ time to resolve the double-black color. So the deletion process in a red-black tree takes $O(\log n)$ time.