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.

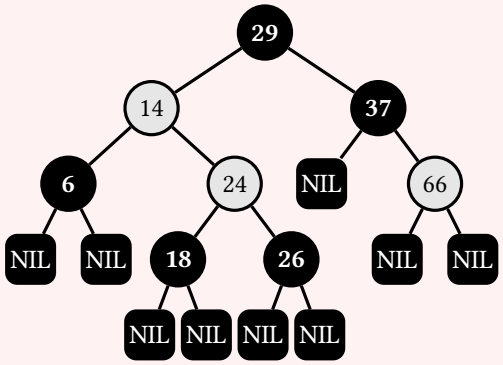


Figure 1.1: A Red Black Tree

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.

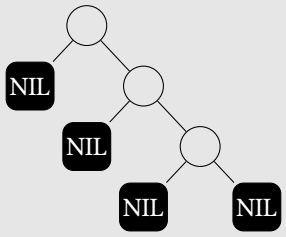
Lemma 1.2

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:

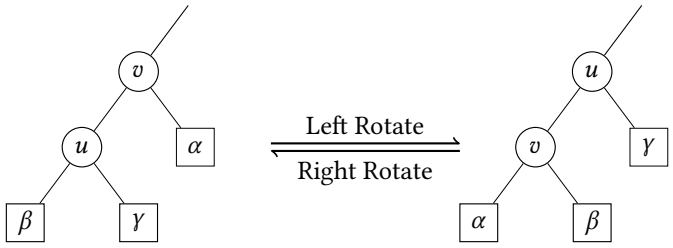


Figure 1.2: Left and Right rotate about $u - v$

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
```

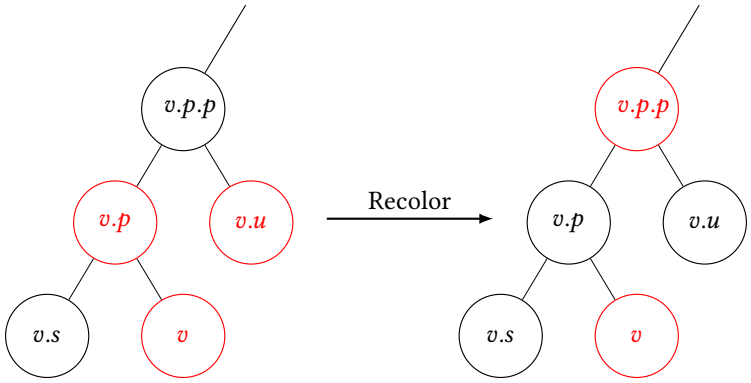
Both Left-Rotate and Right-Rotate take $O(1)$ time. Only some constantly many pointers are changed by rotation all other attributes in a node remain the same.

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 and then use rotations to increase the height of the node at each iteration until the properties of a red-black tree are satisfied.

Let the node added is v. We define the attribute *uncle* which is basically sibling of the parent. Since at the time of addition we coloring the node red it is not hampering the number of black nodes in the path from root to any leaf. So only time it violates the properties of a red-black tree is when the parent of the node is red. 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.