# Red-Black Tree

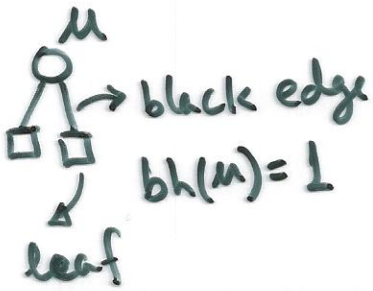
- This is a binary tree where height is balanced and thus it provides efficient search.
- We need to maintain the balance during insertions and deletions.

## Properties of RB tree.

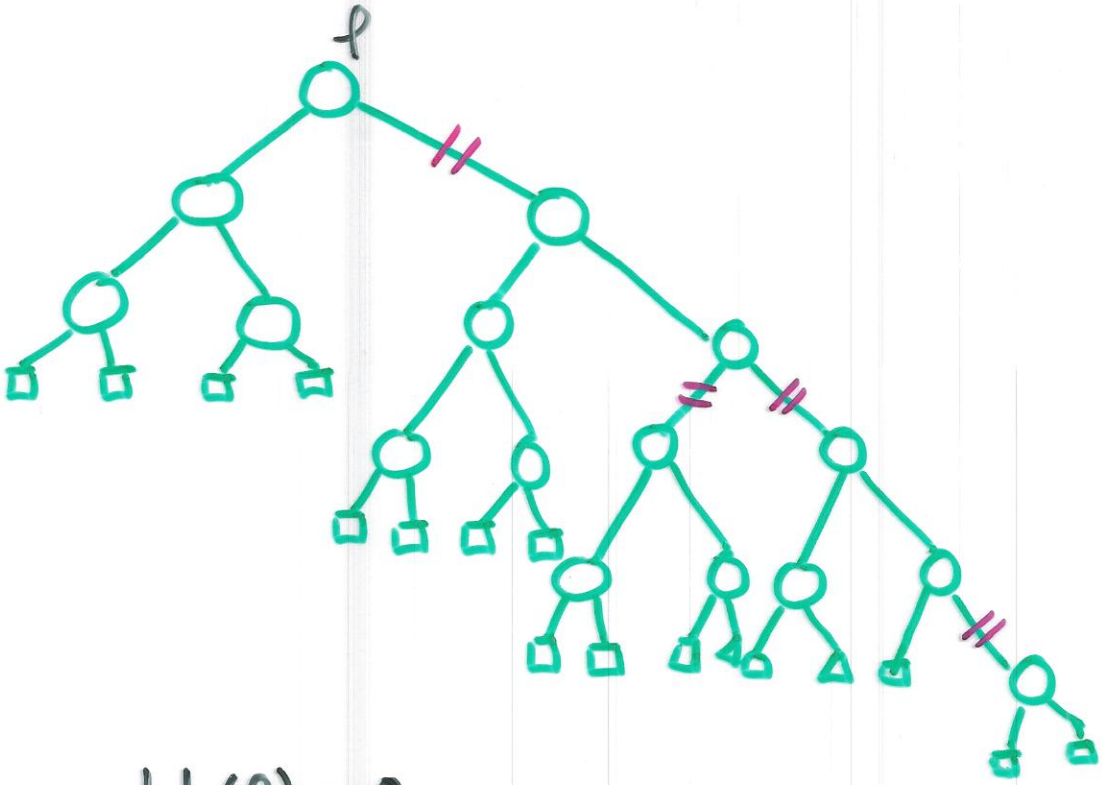
A binary tree where each edge is colored either red or black. One may store a bit in one of the nodes (lower one) to indicate the color.

## Red-black property.

1. Every edge to a leaf is black
2. No (downward) path has 2 consecutive red edges.
3. Every path from a node $u$ to a leaf has the same # of black edges, the <u>black-height</u> $bh(u)$.



**Example.**



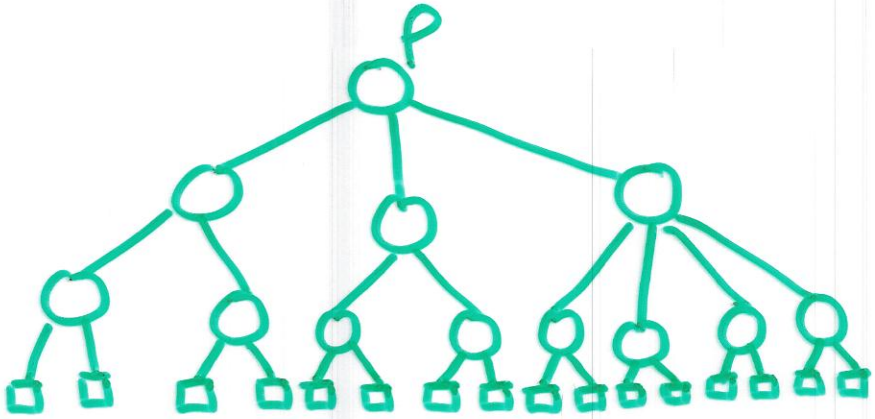

$bh(p) = 3$

$h(p) = 6$

Longest path is at most twice as long as the shortest path from the root to a leaf.

**Claim.** A red-black tree with $n$ internal nodes has height at most $2\log(n+1)$.

**Proof.** The tree has $n+1$ leaves. Now contract red edges, that is, identify their respective 2 nodes. The example red-black tree becomes




Every leaf has depth $bh(p)$.
Every remaining interior node has at least 2 children.
Thus, $2^{bh(p)} \leq n+1$

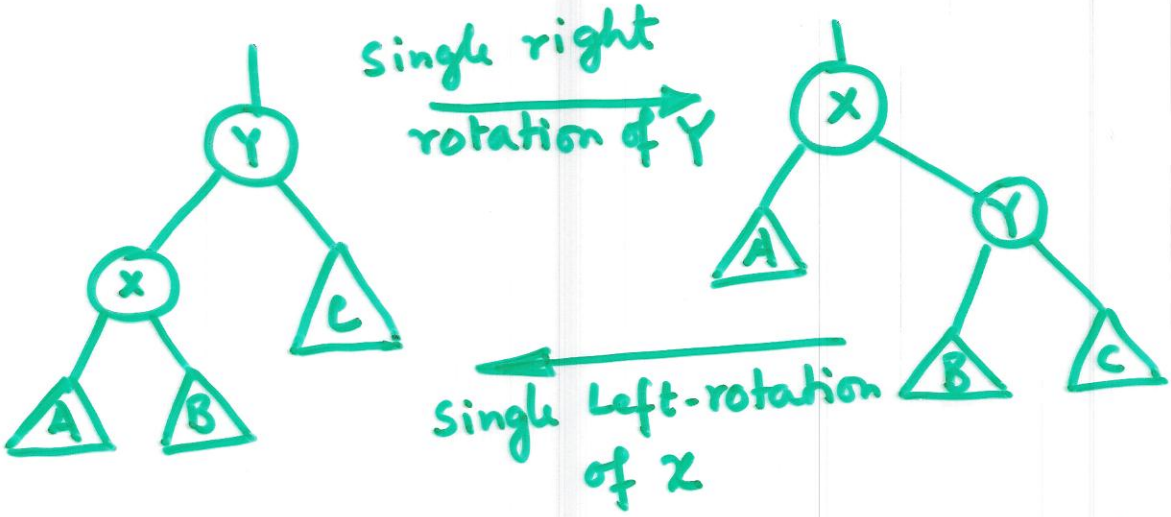
or, $bh(p) \leq \log(n+1)$

or $h(p) \leq 2\,bh(p) \leq 2\log(n+1)$.

The main tool used to balance the tree is rotation.

## Rotations.

A rotation is a local restructuring operation designed to improve the balance.

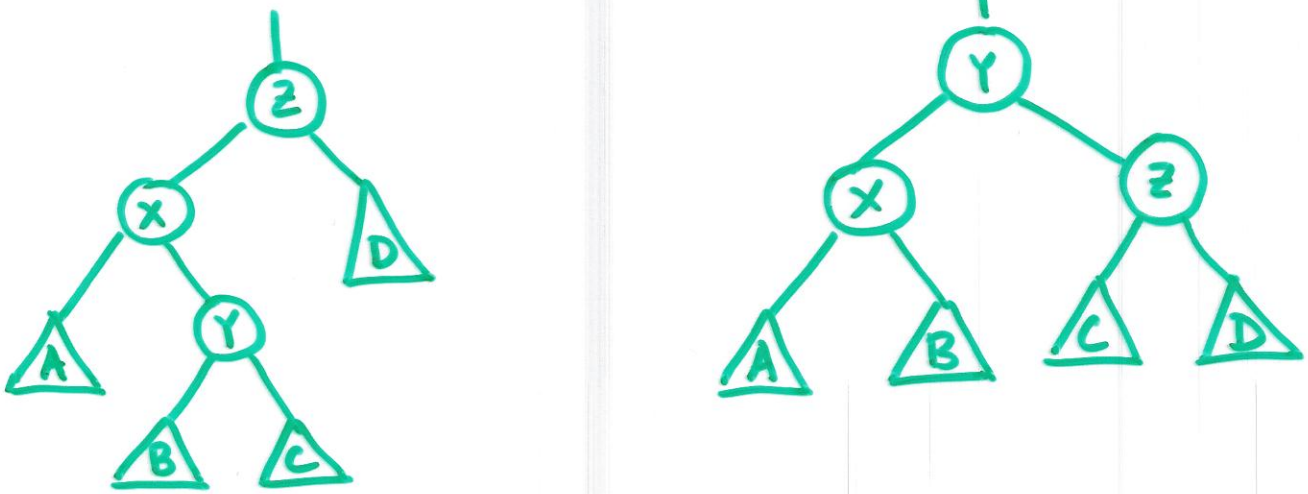



Important: A rotation does not change the inorder sequence.

**Procedure** Left-Rotate (p, x : ↑node) [p: root, assume x.r ≠ nil]

```
y := x.r ; x.r := y.l ;
if y.l ≠ nil then y.l.p := x endif;
y.p := x.p;
if x.p = nil then p := y
             else if x = x.p.l then
                      x.p.l := y
                  else x.p.r := y
                  endif
endif
y.l := x ; x.p := y.
```

There is also a composite type of rotation:

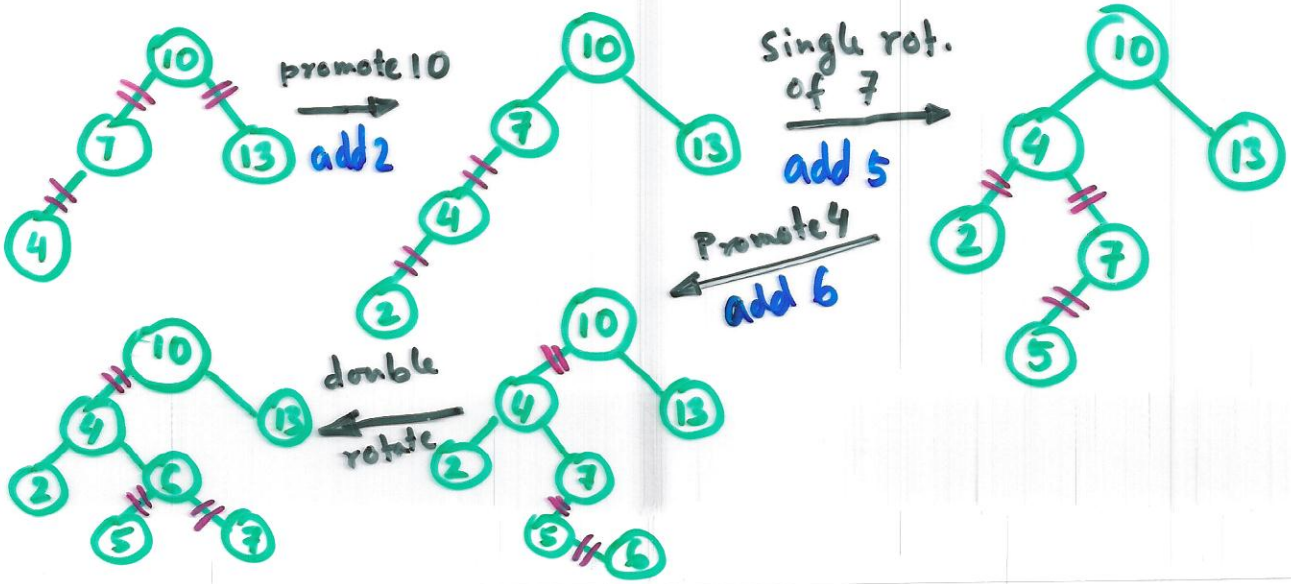



1. Single left rotate x
2. Single right rotate z

An example. We look at the operations for an example first to get an idea.

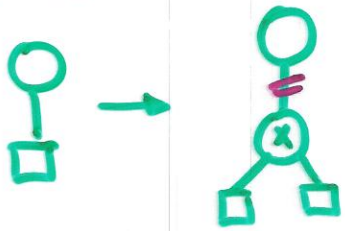
Sequence: 10, 7, 13, 4, 2, 5, 6.

add 10, 7, 13, 4



## Insertion.

First we add the new key $x$ by replacing a proper leaf as for binary search tree.
Color the incoming edge (from parent) red.
Then, adjust color and structure
at $v := x.p$




## Invariant.

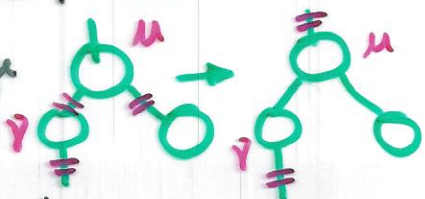
1. if $v$ has a red incoming edge and a red outgoing edge then this is the only violation of the red-black tree property.

2. If $v$ has a red incoming edge then it has exactly one red outgoing edge; otherwise there are one or two outgoing edges.

**Case 1.** Incoming edge of $v$ is black: Done

**Case 2.** Incoming edge of $v$ is red. Set $u := v.p$

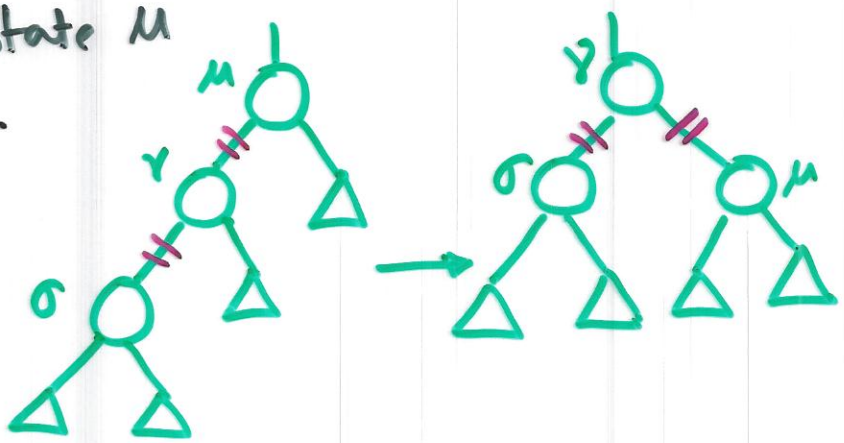
**Case 2.1** Both outgoing edge of $u$ are red;
promote $u$;
if $u.p \neq nil$ then $v := u.p$
and recurse for $v$
endif



**Case 2.2** $v$ is left child of $u$, and left outgoing edge of $v$ is red.

Single rotate $u$ to right.

Done.

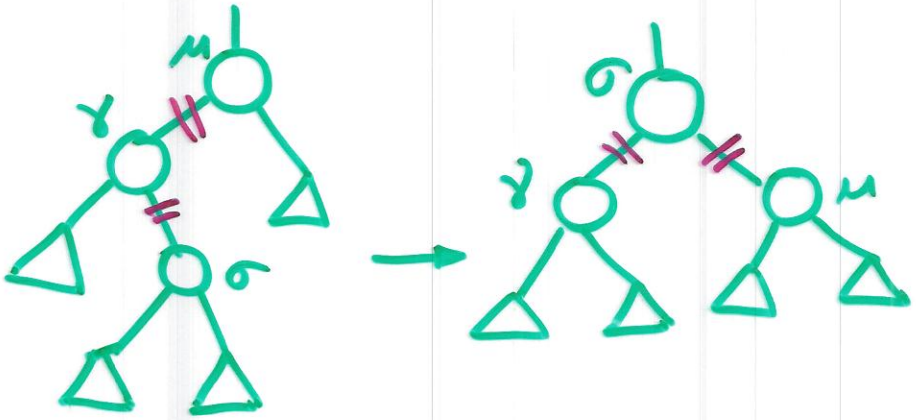



(There is a symmetric right-right case)

**Case 2.3** $v$ is left child of $u$; and right outgoing edge of $v$ is red.

Double rotate $u$ to right

Done.




(There is a symmetric right-left case).

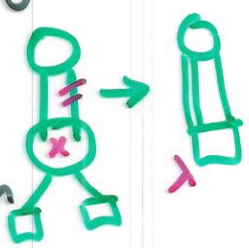
Observe that invariants are maintained and then at most 2 rotations needed.

**Deletions.** First find the node $x$ that stores the item that needs to be deleted.

By substituting with successor or predecessor we can assume that $x$ has 2 leaves as children.

Caveat: you may need to perform several successor or predecessor operation. For example, if $x$ has a right child, do successor and repeat if that successor($x$) has a right child .... and so on. Successor($x$) cannot have left child. If $x$ had a left child but no right child, do a predecessor.....

Since $x$ has 2 leaves, we can replace $x$ by a leaf $\lambda$. If the incoming edge of $x$ is red then that of $\lambda$ should be black. If the incoming edge of $x$ is black then we have a problem and need to restructure. Make the incoming edge as "double black". Start restructing with $Y := \lambda$.



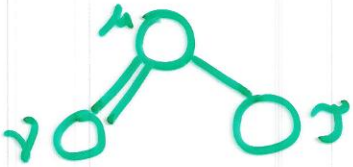
**Invariant.** If the incoming edge of $v$ is black then we have a valid red-black tree; otherwise, the incoming double-black edge is the only violation of the red-black property.

**Case 1.** Incoming edge of $v$ is black. Done.

**Case 2.** Incoming edge of $v$ is double-black. $u := v.p$; $x$ is sibling of $v$.

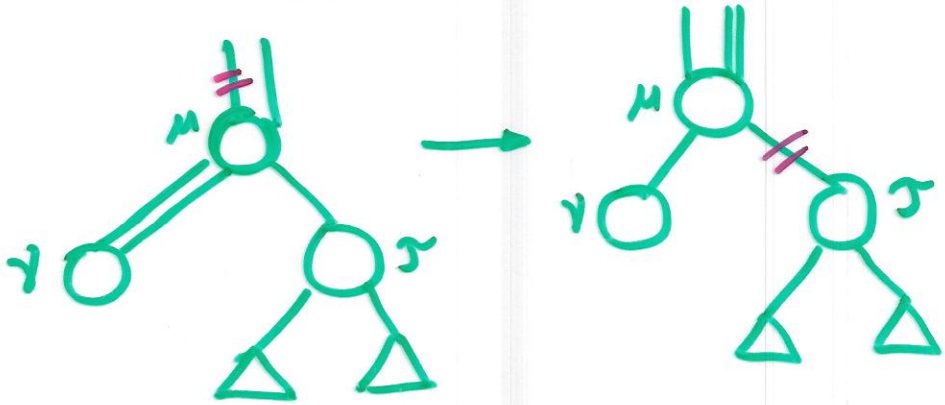
**Case 2.1** Edge from $u$ to $x$ is black.

(In this case $x$ is not a leaf): otherwise, $u$ will not have same black-height on both sides.

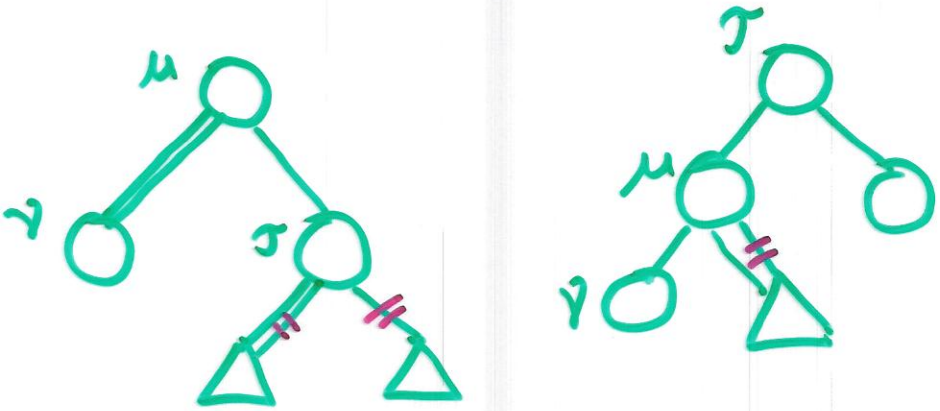



**Case 2.1.1** Both outgoing edges of $x$ are black

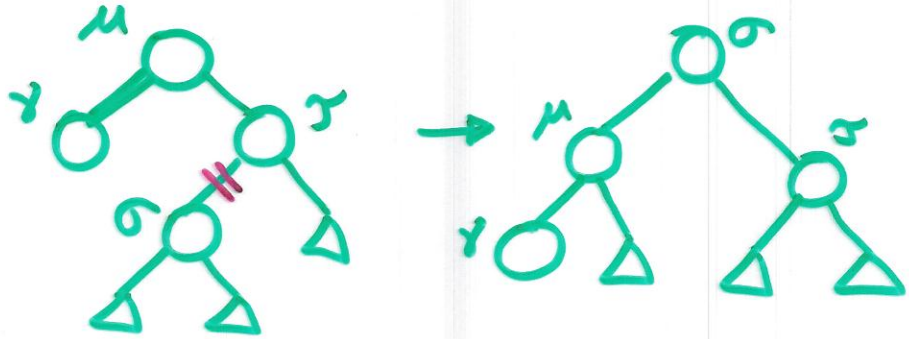
Demote $u$; recurse for $v := u$



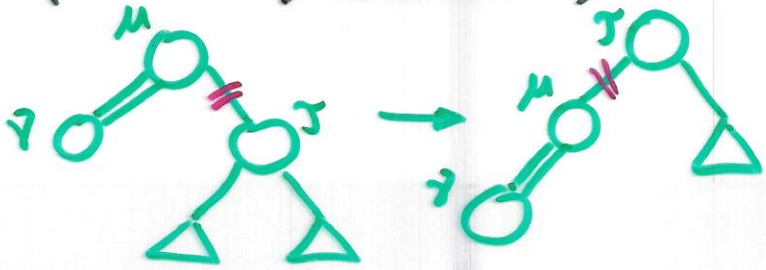
**Case 2.1.2** $v$ is left child of $u$ and right outgoing edge of $x$ is red.
Single rotate $u$ to left. Done.




**Case 2.1.3** $v$ is left child of $u$ and left outgoing edge of $x$ is red, the other one is black. Double rotate $u$ to left. Done.




**Case 2.2** Edge from $u$ to $x$ is red, assume $v$ is left child of $u$. Single rotate $u$ to left, recurse for $v$.
(Next step case 2.1, terminates)



Summary:

A red-black tree supports operations search, minimum, maximum, successor, predecessor, insertion, deletion in time $O(\log n)$ each. A single insertion or deletion requires at most 3 rotations (only during promote or demote recursions happen that do not involve any rotations).