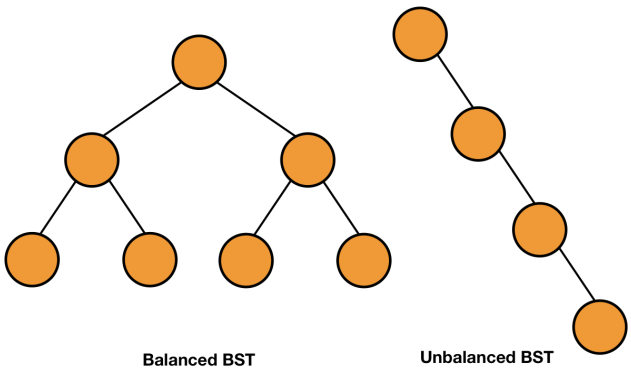
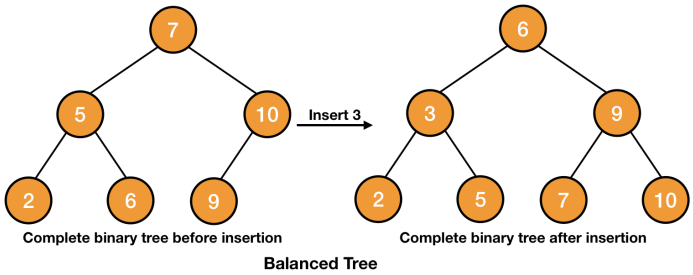
[redacted] (/)

Red-Black Trees

As stated in the previous chapter, this chapter is all about getting a balanced binary search tree. But before going to that, let's discuss when exactly we say a BST to be balanced.

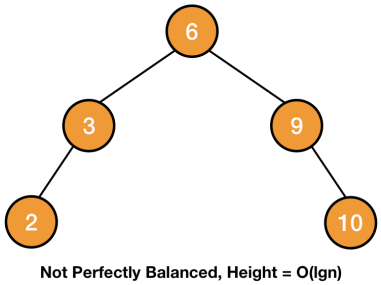


A "**perfect balanced**" binary search tree is basically a complete tree and we should also get a complete tree after any operations like `SEARCH`, `INSERT`, `DELETE`, etc. In this case, the height of the tree will be $O(\lg n)$ and thus, these basics operations can be performed in $O(\lg n)$ worst-case time as h is $O(\lg n)$.



But the thing is we can't rebuild the entire tree as a complete tree after inserting or deleting any node because the building process will be very expensive and then there won't be any use of inserting or deleting in $O(\lg n)$ time.

However, even if a tree is not perfectly balanced but pretty good balanced (little out of balance) can have its height $O(\lg n)$. So, we can still perform basic operations in $O(\lg n)$ worst-case time. And the benefit is that we can maintain that balance without reconstructing the entire tree.

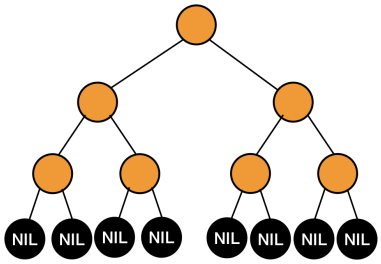


Red-black trees, AVL trees are an example of such trees which use some set of rules which ensure that they are balanced and we will prove that in both trees, the height of the tree is $O(\lg n)$. And we will also see that maintaining those rules after insertion or deletion doesn't take much time which will affect the worst-case running time of $O(\lg n)$ of these operations. So, let's learn about red-black trees in this chapter.

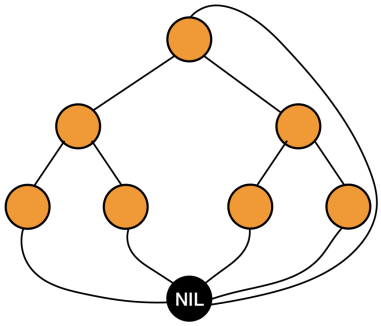
Properties of Red-Black Trees

As stated above, a red-black tree ensures that its height is $O(\lg n)$ by following some properties, which are:

1. Every node is colored either red or black.
2. Root of the tree is black.
3. All leaves are black.
4. Both children of a red node are black i.e., there can't be consecutive red nodes.
5. All the simple paths from a node to descendant leaves contain the same number of black nodes.



Since all the leaves are black, we have used blank nodes or NIL for them as shown in the above picture. We can also use only one node to represent all NIL with black color and any arbitrary values for the parent, data, left, right, etc. as shown below.



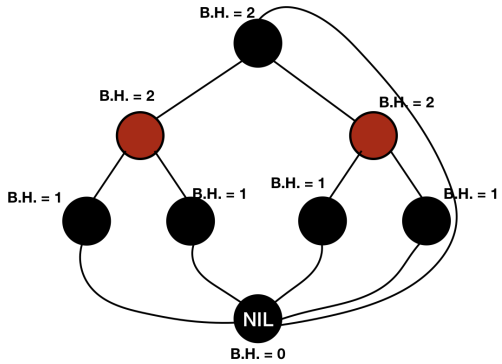
So instead of *NULL*, we are using an ordinary node to represent (NIL) to represent it. This technique will save a lot of space for us.

Black Height of Red-Black Tree

Black height is an important term used with red-black trees. It is the number of black nodes on any simple path from a node *x* (not including it) to a leaf. Black height of any node *x* is represented by $bh(x)$.

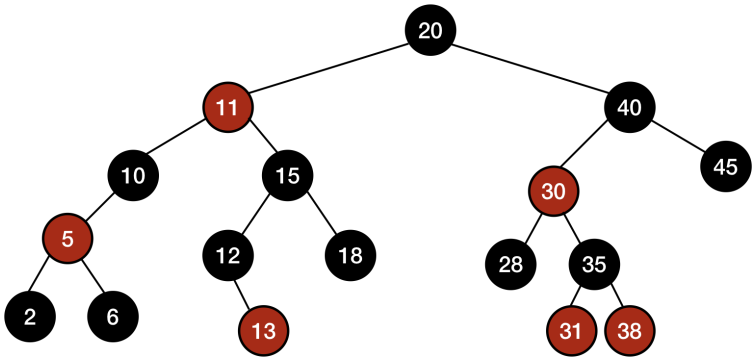
According to property 5, the number of black nodes from a node to any leaf is the same. Thus, the black height of any node counted on any path to any leaf will be unique.

Look at the picture given below with black height of nodes



Black height of the leaf (NIL) is 0 because we exclude the node for which we are counting the black height. Root has a black height of 2 because there 2 black nodes (excluding the root itself) on a path from the root to leaf.

Let's look at one more picture of a red-black tree.



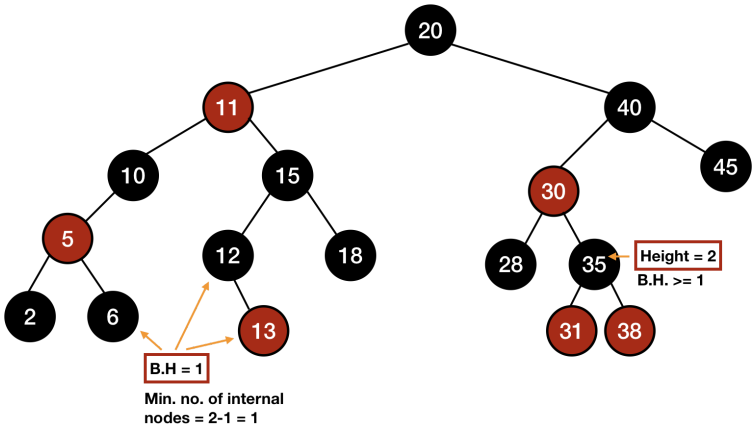
We have omitted the leaves while representing the above tree and we are going to follow the same pattern in this entire chapter. You should keep in mind that there is a *NIL* node representing the leaves in each example.

Proof of height of red-black tree is $O(\lg(n))$

A binary search tree following the above 5 properties is a red-black tree. We also told that basic operation of a binary search tree can be done in $O(\lg n)$ worst-case time on a red-black tree. To prove this, we will first prove that **a binary search tree following the above properties (thus, a red-black tree) with n internal nodes can have a maximum height of $2 \lg (n + 1)$.**

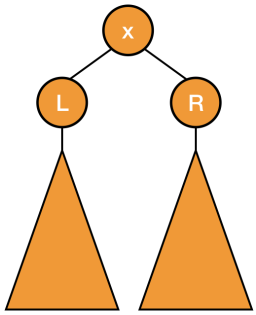
In order to do so, we need to prove the following statements first:

1. A subtree rooted at any node *x* has **at least** $2^{bh(x)} - 1$ internal nodes.
2. Any node *x* with height $h(x)$ has $bh(x) \geq \frac{h(x)}{2}$.

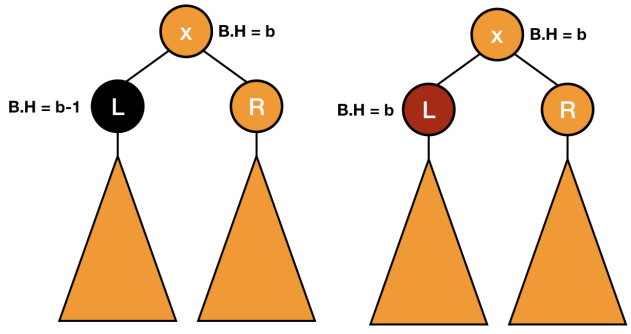


We are going to prove the first statement by the method of induction. The base case will be when *x* is 0 i.e., *x* is a leaf. According to the statement, number of internal nodes are $2^0 - 1 = 0$. Since *x* is a leaf, this statement is true in the base case.

Now, consider a node *x* with two children *l* and *r*.



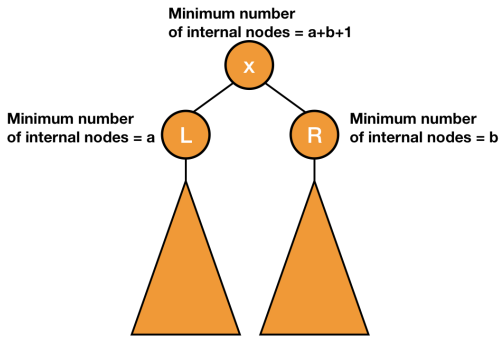
Let $bh(x) = b$. Now if the color of the child is red, then its black height will also be b. However, if the color of the child is black, then its black height will be $b - 1$.



According to the inductive hypothesis, child must have at least $2^{b-1} - 1 = 2^{bh(x)-1} - 1$ internal nodes.

We have assumed that the inductive hypothesis is true for the child, now we need to show that it is also true for the parent i.e., node *x* and hence completing the proof.

The node *x* must have at least 1 + least number of internal nodes that can be present on the right child + least number of internal nodes that can be present on the left child.



i.e.,

$$2^{bh(l)-1} + 2^{bh(r)-1} + 1$$

Internal nodes of $x \geq 2^{bh(x)-1} + 2^{bh(x)-1} + 1$

or, Internal nodes of $x \geq 2 * (2^{bh(x)-1}) - 1$

or, Internal nodes of $x \geq 2^{bh(x)} - 1$

We assumed it to be true for the child and it is also true for the node *x*, so the statement is proved.

Let's prove the second statement.

Since the leaves are black and there can't be two consecutive red nodes, so the number of red nodes can't exceed the number of black nodes on any simple path from a node to a leaf. Therefore, we can also say that at least half of the nodes on any simple path from a root to a leaf, not including the node, must be black.

It means that $bh(x) \geq \frac{h(x)}{2}$

Thus, the second statement is also true.

Acoording to the second statement, $bh(root) \geq \frac{h}{2}$, where h is the height of the tree.

Using statement 1, $n \geq 2^{bh(root)} - 1$

or, $n \geq 2^{\frac{h}{2}} - 1$ $(bh(root) \geq \frac{h}{2})$

or, $n + 1 \geq 2^{\frac{h}{2}}$

Taking log on both sides,

$\lg(n+1) \geq \frac{h}{2}$

or, $h \leq 2\lg(n+1)$

Thus, the height of a red-black tree is $O(\lg n)$.

</>
You can see this (https://cs.stackexchange.com/questions/54157/can-i-simplify-logn1-before-showing-that-it-is-in-ologn/54159) to prove $O(\lg(n+1)) = O(\lg n)$.

From the previous chapter, we know that basic operations require $O(h)$ time in a binary search tree and we have proved that the height*h* is $O(\lg n)$. Thus, all those operations can be done in $O(\lg n)$. time in a red-black tree.

We need to maintain the properties of a red-black tree while inserting or deleting any node. We will study it in the next chapter.

?

> If I have seen further it is by standing on the shoulders of Giants
>
> - Isaac Newton

PREV (/course/data-structures-binary-search-trees/) (/course/data-structures-red-black-trees-insertion/) NEXT