RED-BLACK TREES in C

A- Definition:

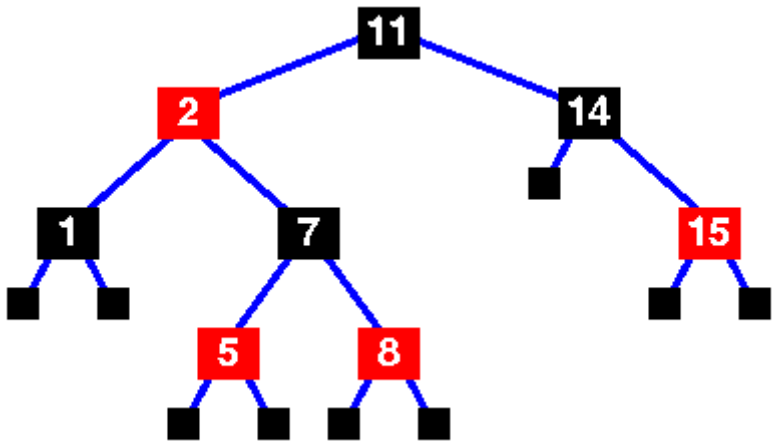
- A **red-black tree** is a **binary search tree** in which each node has an additional factor which is the colour: red or black.
- The colour bits ensure that the tree stays balanced after performing some operations.
- A binary search tree is an ordered binary tree where all the nodes in the left sub-tree are less than then the value of the root node, while the elements on the right sub-tree have values greater or equal than that of the root.

We can also say that a red-black tree is **a self-balancing** binary search tree in a way that any node in this tree automatically keeps its height (total number of nodes on the path from the root to the deepest node in the tree) small after performing insertions and deletions as for the case of the AVL trees .

B- Properties:

In addition of the binary search tree's restrictions, the red-black tree has the following characteristics:

1- The colour of the node is either red or black;
2- The colour of the root node is always black;
3- All the leaf nodes are black and store no data;
4- Every red node has both children coloured black;
5- They are no adjacent red nodes;(in other words, a red node cannot have a red child or a red parent)
6- Every simple path from a given node to any descendant leaf has an equal number of black nodes;



Lemma:

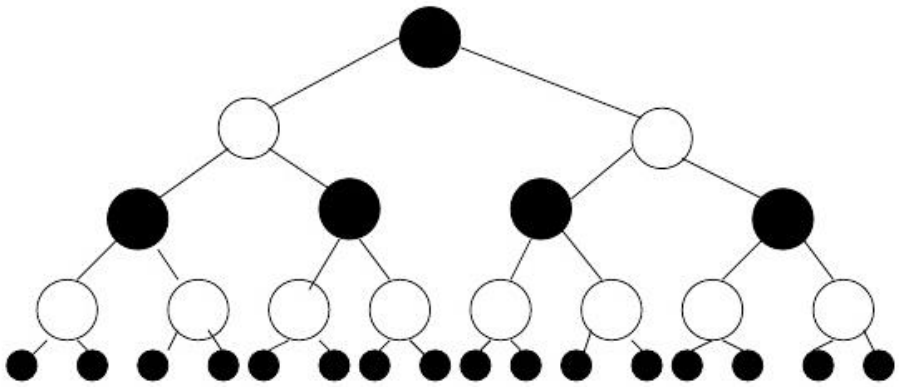
The longest path from the root to any leaf node is no longer than twice as long as the shortest path from the root to any other leaf. (The example above illustrates this lemma).

Explanation:

We previously said that both children of any red node are black, thus in the shortest path, at most, every node is black and the longest one will have a red and a black one alternatively. And since every simple path from a node to a leaf contains the same number of black nodes (property 6), we get that no path is more than twice as any other path.

Black height of a RBT:

It's the number of black nodes on the path from the root the leafs that are also counted as black nodes. For example, if we count the number of black nodes , in the second level, there are four black nodes, and the leafs that are formed in the last level , so the black height of our tree in this example is 2;



A valid Red-Black Tree

Black-Height = 2

Declaring a red-black tree in C:

```
Struct node {
        Int data;
        Char colour;
        Struct node *left;
        Struct node *right;
        Struct node *parent ;};
```

B- Operations on a RBT:

Every red-black tree is a special binary search tree. Some operations require no changes on the tree as for traversing/searching its elements. However, insertion and deletion do change the structure of our tree, thus violate its properties. So to rebalance the tree, there are two tools that are used: recoloring or rotating.

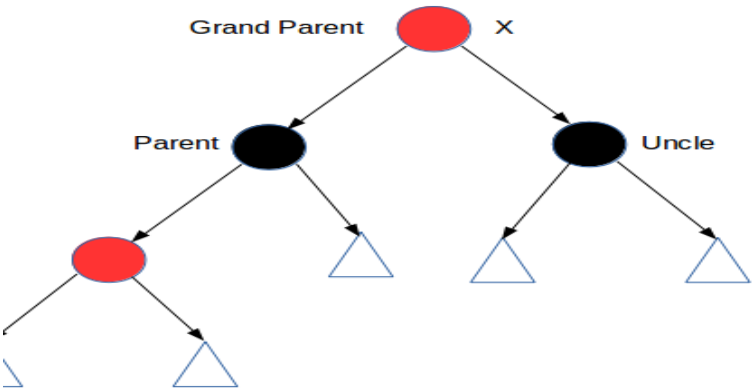
1- Searching for a node in a red-black tree:

Searching for a certain node means checking if that node exists in the tree or not. Simply, to search for a value in a red black tree, we follow the same structure as for the BST. In other words, we start from the root, if the value we're searching for is less than that of the root, we should look for it recursively in the left-sub tree, else, we search in the right one.

2- Inserting a node in a red-black tree:

In a BST, the new node is always added as a leaf containing the data of the inserted element. However, in a RBT, we said that the leafs are black nodes that contain only the Null element, so to insert a node, we add it as a red interior parent node that contains the data of the new node with two black leafs that contain the Null. Now we should check if the other properties are fulfilled as well. But before studying the cases, let's introduce some important terms that will be used from now on:

- Grandparent node (G) of a node (N) refers to the parent of (N)'s parent, as a normal human family tree.
- Uncle node (U) of (N) is the sibling of (N)'s parent as shows the following tree.



The frequent properties that are violated after insertion are the 4th one that states that the children of any red node must be black, and the 6th one that states that the path from any node to its leaf have an equal number of black nodes. Repainting and rotation help restoring our tree correctly.

Now we have 5 cases to recover:

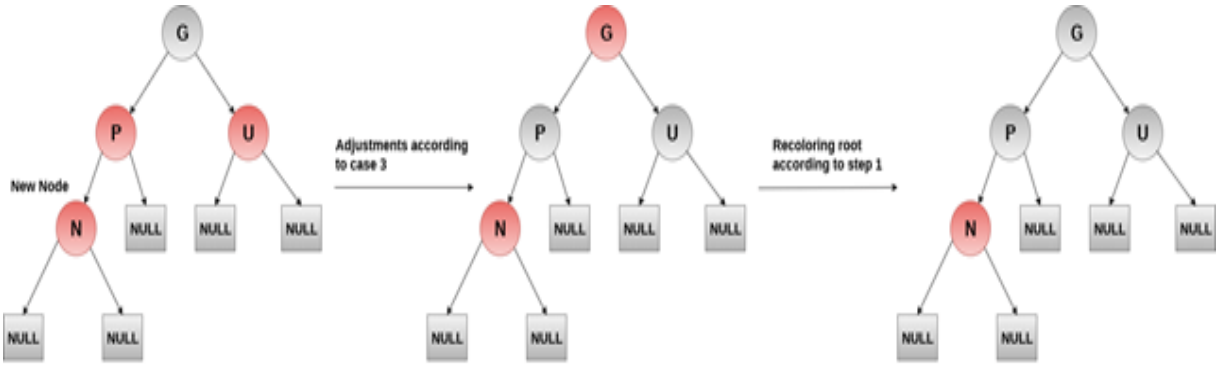
Case 1: The node is added as the root of the RBT:

We said that the added node is red but the root is always black, To solve it we should repaint the node black by modifying its colour field; (only a O (1) is used to change the colours);

Case 2: ***the node's parent is black:***
In this case, both the 4th and 6th properties are violated. The added node as we said has two black leafs, but since N is red the path from the node through its children will have the same number of black nodes.

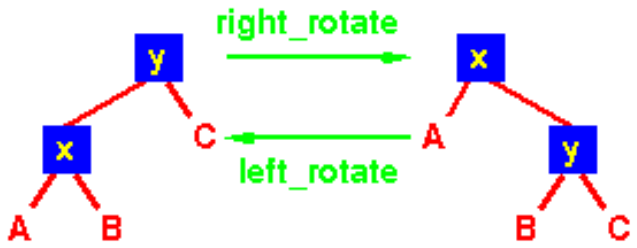
Assuming that N has a parent and an uncle we have:

Case 3: ***both the node's P and U are red:***
In this case, the path from any node to its leafs won't have an equal number of black nodes. Repainting U and P black will make the 6th property valid and colouring the G red will validate the 4th property. Now, let's consider the case where G is a root, the 2nd property won't work since G in our case is red, and changing its colour to black will violate the 4th property again, so with this, we're back to the first case, thus to restore it , we apply the procedure recursively on G. The case is illustrated in the following example;



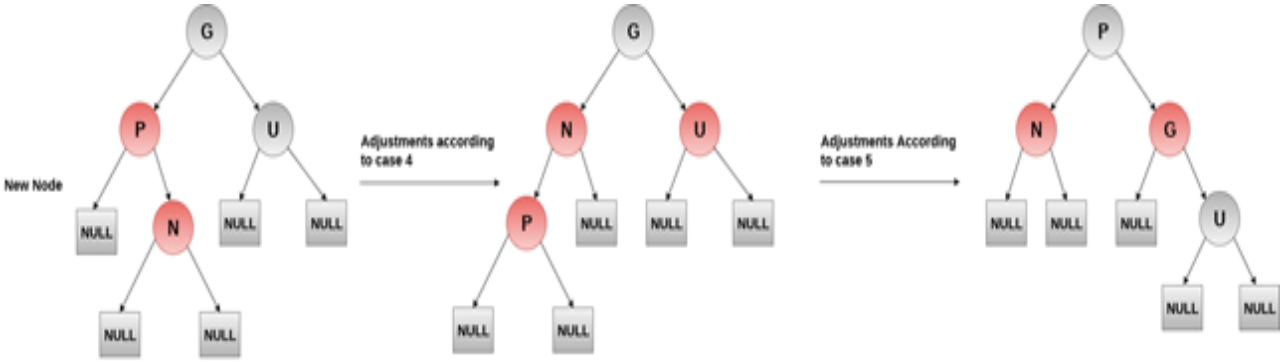
Inserting new node into red black tree
(Case 3: Uncle and Parent both are red)

In the remaining cases, a new operation "rotation" is introduced to make insertion easier and to balance the tree. A rotation is a local operation that preserves the in-order traversal key ordering. (Only O (1) is used to rotate a node). In RBT, we will use the right rotation RR and left rotation LR.



Case 4: ***P is red and U is black and N is the right child of P and the left child of G:***

First, we rotate the tree left (LR) so that the roles of N and P are switched; the right rotation (RR) will be performed if the N is the left chid and P is the right child of G. the following scheme illustrates this case;

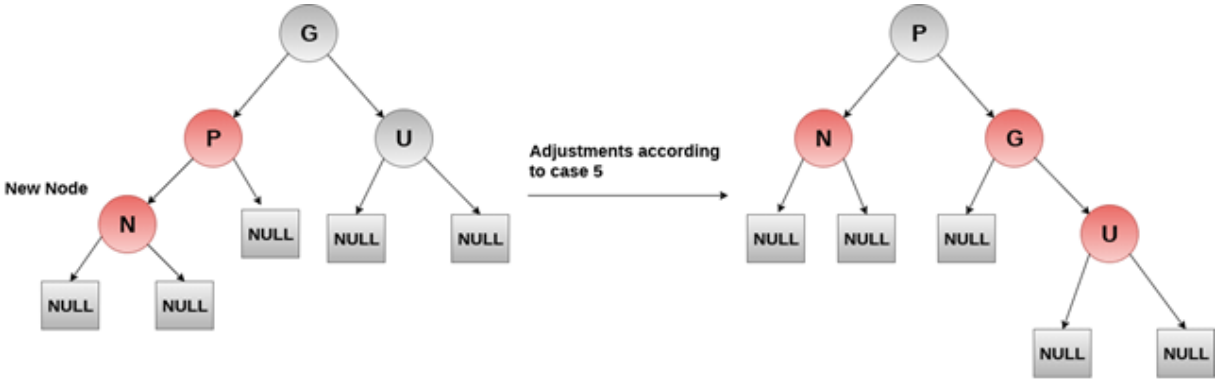


Inserting new node into red black tree

(Case 4: Parent P is red and uncle U is black, N is the right child of P which is the left child of G)

Case 5: P ***is red and U is black and N is the left child of P and P is the left chid of G:***

To solve this, a RR is done on G. P is now the parent of both N and G. we know that both black children have a red parent , thus we should repaint G red and P black to satisfy this 4th property. The following scheme illustrates this case;



Inserting new node into red black tree

(Case 5: Parent P is red and uncle U is black, N is the left child of P which is the left child of G)

2- Deleting a node from a red-black tree:

In insertion, we started by checking the colour of the uncle of the node to deal with the right case, but in deletion as we're going to do now, we will check the colour of the sibling to choose the right case.

In binary search trees, we faced 3 cases; either we delete a leaf (easier case), a node with one child, or a node with two children. For that, we replace the node by either its in-order predecessor (max in the left-sub tree) or by its in-order successor (min in the right-sub tree) and we finally free then node.

The same thing goes for the red black trees. But instead of studying the three cases, we will only consider the second case where the deleted node has only one child. If the deleted node is red we replace it with its black child. But if the deleted node is black having a red child, we first repaint the child black and then we replace the node with this new child.

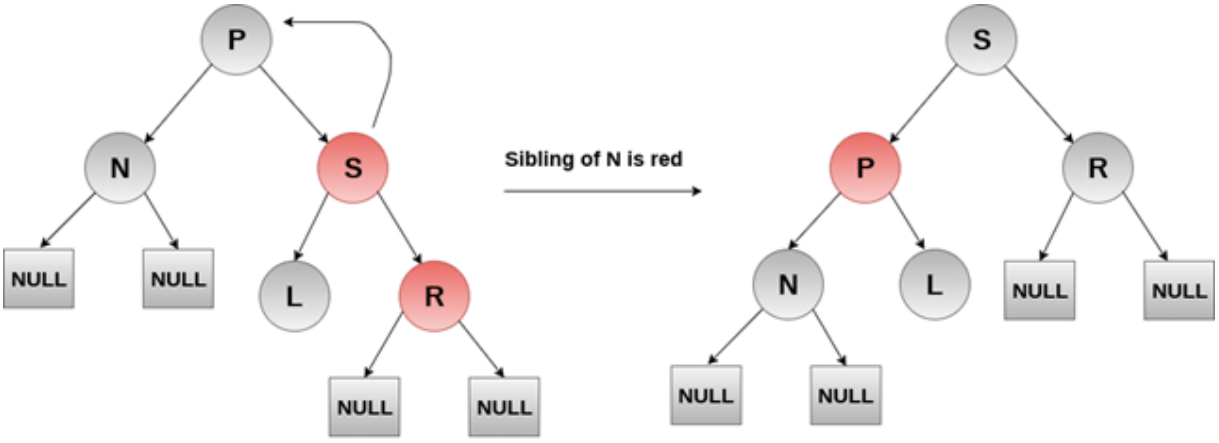
But when both the deleted node and its parent are black, there are several cases to recover:

Case 1*: *N is the new root:

In this case, the new root is black so none of the properties are violated.

***Case 2*: *N' sibling is red*:**

A node' sibling is the node that has the same level and parent. What we should do in this case is to switch the parent and the sibling's colours and then RL the parent so that S becomes the node's grandparent. This is shown in the following example;

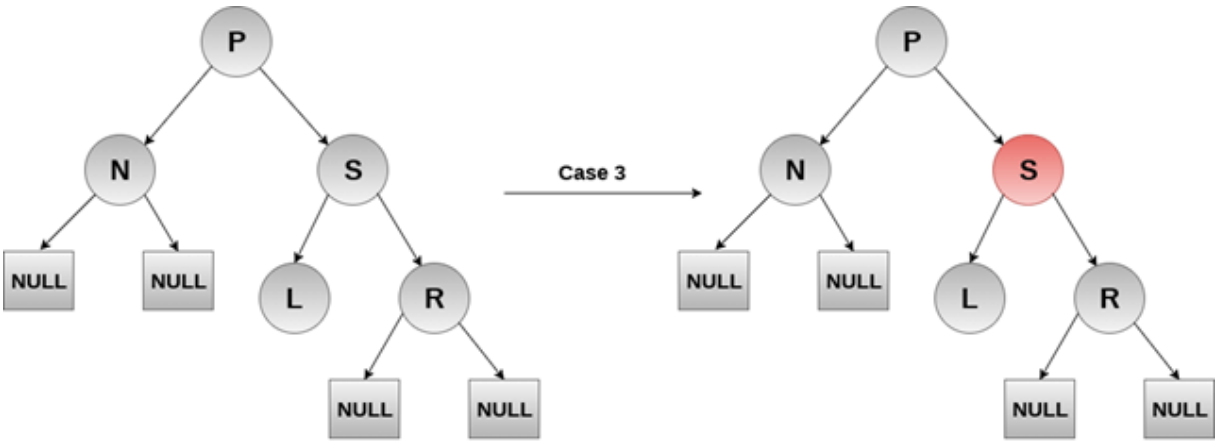


Deletion in Red Black Tree

(Sibling of node is red)

Case 3*: *both S and its children are black and so is P:

In this case, we repaint S red and we perform the rebalancing procedure from the first case on P to apply the 6th property since all the paths going through S will have one less black node. The following scheme illustrates this case;

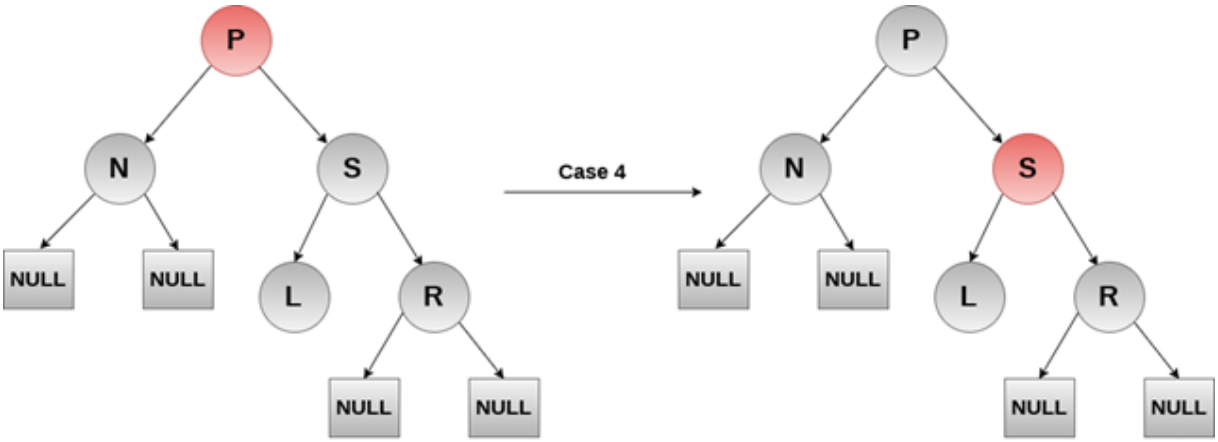


Deletion in Red Black Tree

(Case 3)

Case 4*: *S and its child are black, P is red:

we swap P and S's colours knowing that this won't affect the number of black nodes which should be the same from a node to its leafs. The following scheme illustrates this case;

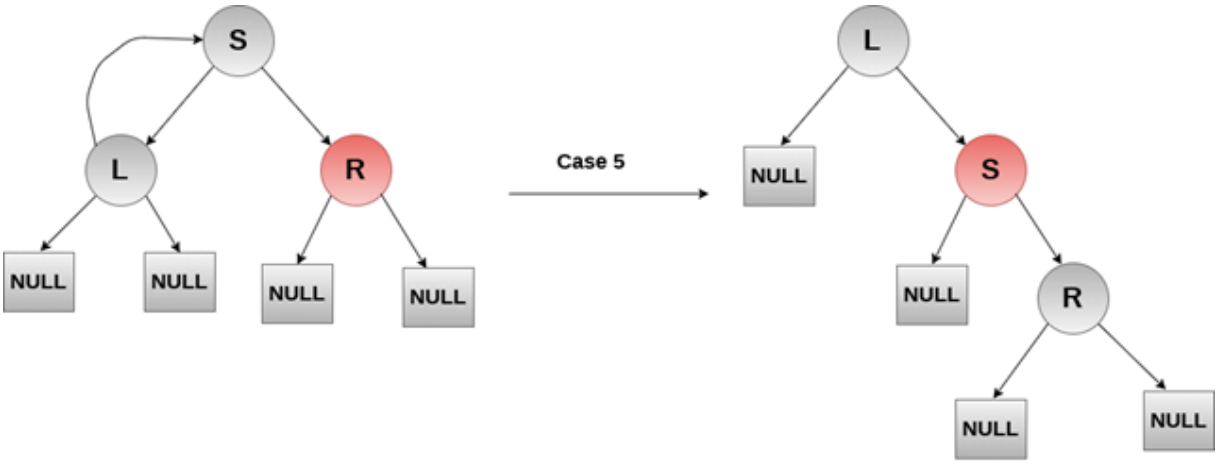


Deletion in Red Black Tree

(Case 4)

Case 5: N is the left child of P and S is black, S's right child is black and its left child is red:

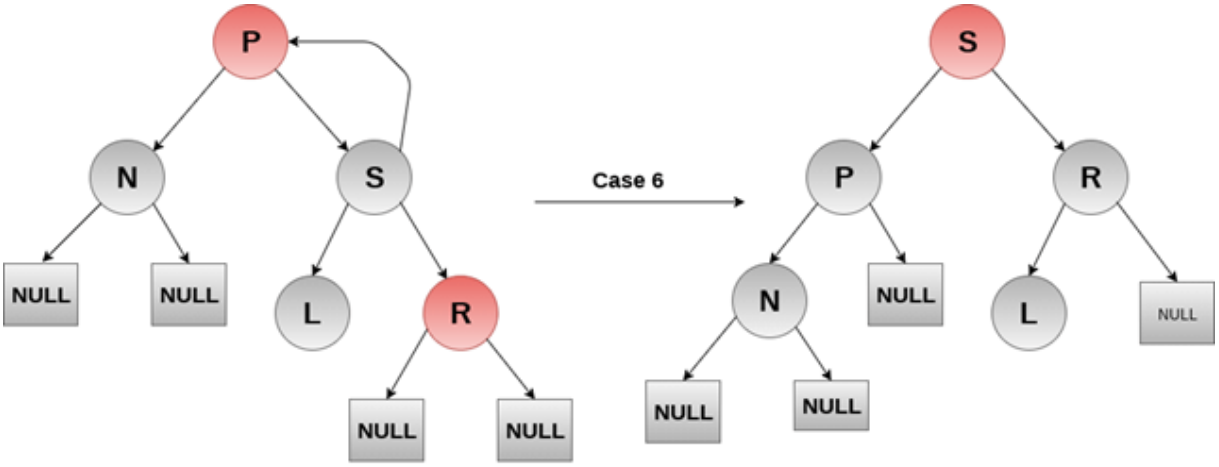
We perform a RR on S so that the left child of the sibling becomes S's parent, and swap the S and its parent's colours. Now the 6th property is fulfilled, but this will lead to the next case. The following scheme illustrates this case;



Deletion in case 5

Case 6: ***N is the left child of P; S is black node having a red right child:***

In this case, we apply a LR on P so that S becomes the P's parent and the right child of S. The following scheme illustrates this case;



Deletion in Red Black Tree

(Case 6)

D- Red-black trees vs. AVL trees:

AVL trees are :	RB trees are :
❖ Easy to balance if only fewer insertion and deletions are done; ❖ Additional factor : weight of the node; ❖ Used most of the times in databases; ❖ Rebalancing (rotation) is done if the weight of two child sub-trees of a node differ by more then 1; ❖ Faster lookups since they are more balanced,	❖ Faster when a lot of insertions or deletions are done; ❖ Additional factor: colour of the node; ❖ Often used in most languages; ❖ Rebalancing (rotation colouring) is done once a property is violated. ❖ AVL trees can be a special RBT if we colour the nodes.

E- Some applications of red-black trees in real life:

As a conclusion, we can say that a red-black tree is valuable in sensitive-time applications such as real-time applications. It's an efficient and balanced

binary search tree that offers a O (log (n)) search, insertion and deletion time where n is the number of nodes. It's mostly used in the process scheduler in Linux as a replacement for the run queues with priorities, and to keep track of the virtual memory segments of a process.

A red-black tree can be also used as a block for building other data structures with a worst-case guarantee to perform those several operations for example computational geometry is based on this tree.

In addition of that, the collection HASHMAP in java8 has been modified so that instead of using linked lists to store data, red black trees are used (O(log(n)) is way better than O(n)).

Reference:

- **https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/**
- **https://walkccc.github.io/CLRS/Chap13/13.1/**
- **https://www.javatpoint.com/red-black-tree**
- **https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html**