Performance of the basic operations (search, insert, delete a node) for a binary search tree with n nodes is proportional to the height of a binary tree h. In the worst case the running time of these operations is linear with respect to the number of nodes of the tree ($= O(n)$) so it is not better than using a linked list with n nodes. Using special techniques for binary search trees as for instance, AVL rotations or red-black trees, guarantees that all basic operations can be done in logarithmic running time $O(\log_2 n)$. These special techniques maintain the height of a binary tree of order $O(\log_2 n)$.

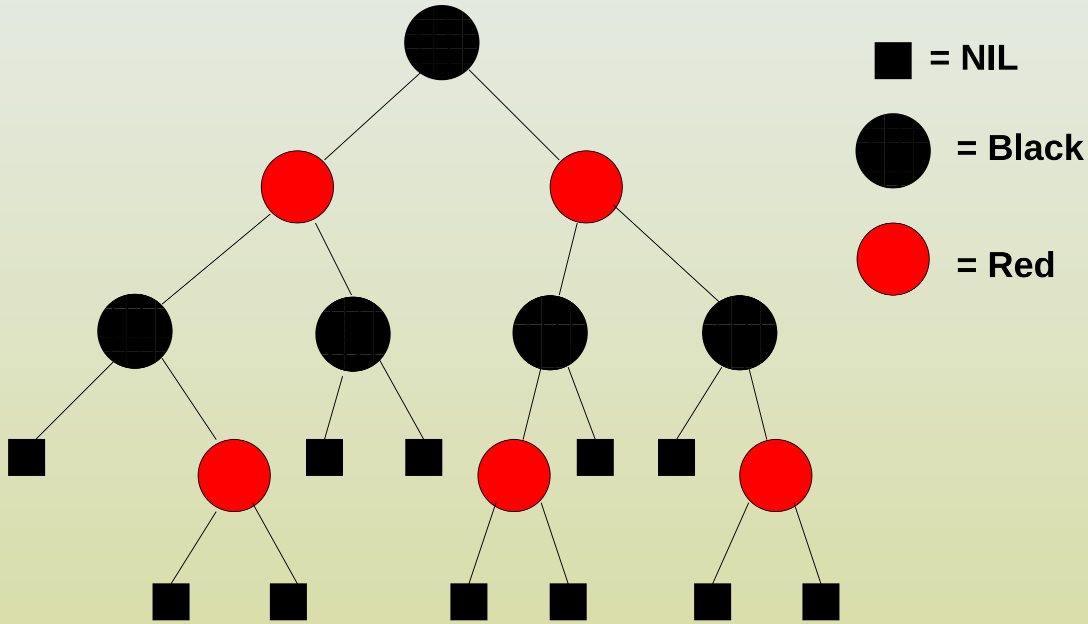
Red-black tree is a binary search tree with the following properties:

Every node in the tree is either red or black.

Every leaf node (NIL) is black.

If a node is red, then both its children are black (there are no two consecutive red nodes in the tree).

Every simple path from a node to a descendant leaf contains the same number of black nodes.



Height of Red-Black Trees

Definition.

Black-height of a node x is a number of black nodes on any path from x (but not including x) to a leaf (NIL node).

Black-height is well defined, because by the property 4 there is the same number of black nodes on any path form a node to a leaf. We denote a black-height of a node x by $bh(x)$. The black-height of a red-black tree is the black-height of its root.

Lemma.

A red-black tree with n internal nodes has height $\Theta(\log_2 n)$.

Proof.

Notation: $bh(x)$ = black-height of x, $h(x)$ = height of the red-black tree rooted at x. Note that $bh(x) \leq h(x)$,
$n(x)$ = number of internal nodes in red-black tree rooted at x. It is not difficult to notice that $bh(x) \geq h/2$ because there cannot be more red nodes than black nodes on any path from the root to a leaf by the property 3.
Now we need to show that the subtree with the root at a node x contains at least $2^{bh(x)} - 1$ internal nodes. We can prove it by induction with respect to $h(x)$.

Basic induction step: If x is a leaf (NIL) then $h(x) = 0$ and such a tree contains at least $2^0 - 1 = 0$ internal nodes.

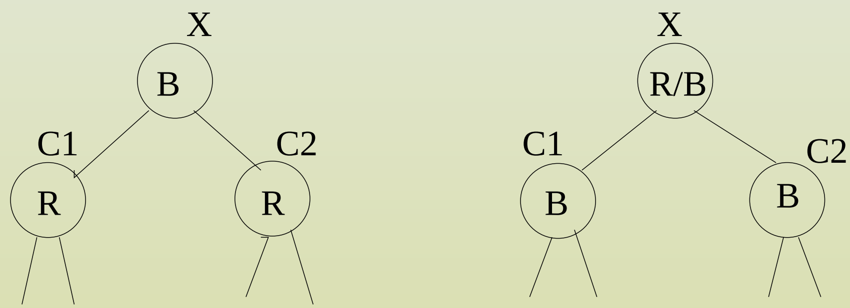
Assume that for $h(y) = k - 1$ then $n(y) \geq 2^{bh(y)} - 1$.

We prove that if $h(x) = k$ then $n(x) \geq 2^{bh(x)} - 1$.

There are two cases to consider:

x is black and its children are red.

x is red or black and its children are black.



The first case.
From the definition of the red-black tree

$bh(c1) = bh(x)$ and $bh(c2) = bh(x)$

From the induction assumption

$n(c1) \geq 2^{bh(x)} - 1$ and $n(c2) \geq 2^{bh(x)} - 1$

Hence

$$n(x) = n(c1) + n(c2) + 1 \geq 2 \times 2^{bh(x)} - 2 + 1 \geq 2^{bh(x)} - 1 \text{ nodes}$$

The second case.

From the definition of the red-black tree

$bh(c1) = bh(x) - 1$ and $bh(c2) = bh(x) - 1$

From the induction assumption
$n(c1) \geq 2^{bh(x)-1} - 1$ and $n(c2) \geq 2^{bh(x)-1} - 1$

Hence

$$n(x) = n(c1) + n(c2) + 1 \geq 2 \times 2^{bh(x)-1} - 2 + 1 = 2^{bh(x)} - 1 \text{ nodes.}$$

This proves that in any case we have at least $2^{bh(x)} - 1$ nodes.

Knowing that we the relation

$$n(x) \geq 2^{bh(x)} - 1,$$

where x is root of the red-black tree. Since we also know that $bh(x) \geq h(x)/2$ the we get

$$n(x) \geq 2^{bh(x)} - 1 \geq 2^{h(x)/2} - 1.$$

After some algebra we finally get

$$h(x) \leq 2 \log_2(n(x) + 1).$$

Since we know that $h(x) \geq \log_2(n(x) + 1)$ (since $n(x) \leq 2^h(x) - 1$) for any binary tree, we get $h(x) = \Theta(\log_2(n(x) + 1)$.

Red-Black Tree Structures in C

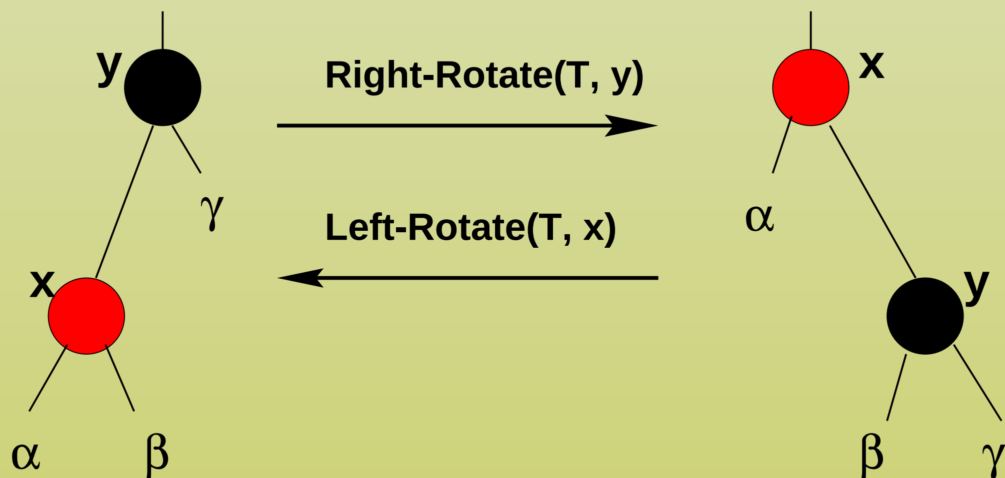
```
typedef enum {red, black} Color;

typedef struct RBTNode {
   RBTNode *left, *right, *parent;
   int key;
   Color color;
   /* other fields if any */
} RBTNode;

typedef struct {
   RBTNode *root;
} RBTree;
```

Rotations

The operation **Tree-Insert**, when run on a red-black tree with n nodes, takes $O(\log_2(n))$ time. This operation may violate the red-black properties. We can restore these properties by changing colors (red $\rightarrow$ black and black $\rightarrow$ red) of some nodes, and by using rotations. Here we describe **Right-Rotate** operation. A rotation is a local operation which preserves the inorder key ordering of a red-black tree. The figure below shows left and right rotations.



Right Rotation

```
/* we assume that y->left != NULL */
void RightRotate(RBTree *T, RBNode *y){
  RBNode *x;
x = y->left;
  y->left = x->right;/* turn x's right subtree into y's left subtree */
  if (x->right != NULL) x->right->parent = y;
  x->parent = y->parent;   /* link y's parent to x */
  if (y->parent == NULL)
    T->root = x;
 else if (y == y->parent->right)
        y->parent->right = x;
      else y->parent->left = x;
 x->right = y;
 y->parent = x;
}
```

The code for LeftRotate is similar (you need to change left → right and right → left). Both the rotation operations run in $O(1)$ time. They do not copy any structures, only pointers are involved.

Inserting a Node

It is possible to insert a node in a red-black tree in $O(\log_2(n))$ time, where n is the number of nodes of the red-black tree. To establish red-black properties after inserting a new node we need to re-color nodes and perform rotations. The three main cases in the code are discussed below. Note that `RBTNode` is `BSTNode` with added `color` field.

```
void RBInsert(RBTree *T, RBTNode *x)
{
RBTNode *y;
TreeInsert(T, x);  /* ordinary BST insertion */
x->color = red;
while (x != T->root && x->parent->color == red) {
  if (x->parent == x->parent->parent->left) {
   y = x->parent->parent->right;
   if (y != NULL && y->color == red) {   /* case 1 */
```

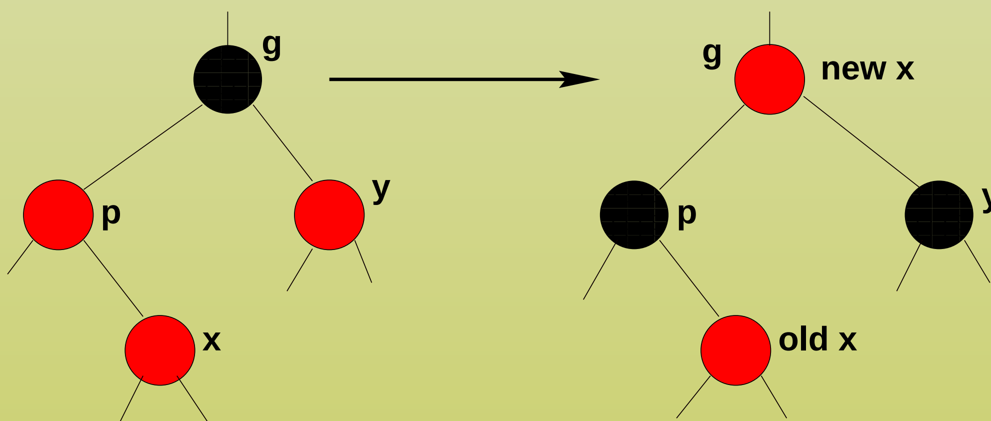
```
            x->parent->color = black;
            y->color = black;
            x->parent->parent->color = red;
            x = x->parent->parent;
    } else { /* case 2 & 3 */
        if (x == x->parent->right) { /* case 2 */
            x = x->parent;
            LeftRotate(T, x);
        } /* case 3 */
            x->parent->color = black;
            x->parent->parent->color = red;
            RightRotate(T, x->parent->parent);
        }
    } else { /* see the next slide */
```

```
} else { /* x->parent == x->parent->parent->right */
         y = x->parent->parent->left;
         if (y != NULL && y->color == red) {  /* case 1 */
            x->parent->color = black;
            y->color = black;
            x->parent->parent->color = red;
            x = x->parent->parent;
         } else { /* case 2 & 3 */
              if (x == x->parent->left) {  /* case 2 */
              x = x->parent;
              RightRotate(T, x);   }  /* case 3 */
          x->parent->color = black;
          x->parent->parent->color = red;
          LeftRotate(T, x->parent->parent);  }
         }   } /* end of while */
  T->root->color = black;
}
```

The inserted node `x` is always colored red. Note that the root of a red-black tree is always black (see the last line of the code). `TreeInsert` may not preserve red-black properties, therefore we need to restore them. It is easy to notice that property 1, 2 and 4 are preserved when we add a red node. Only property 3 may be violated. And this happens only when the parent of the added node `x` is also red (when the parent is black we do nothing). The goal of the `while` loop is to move violation of property 3 up while maintaining property 4.

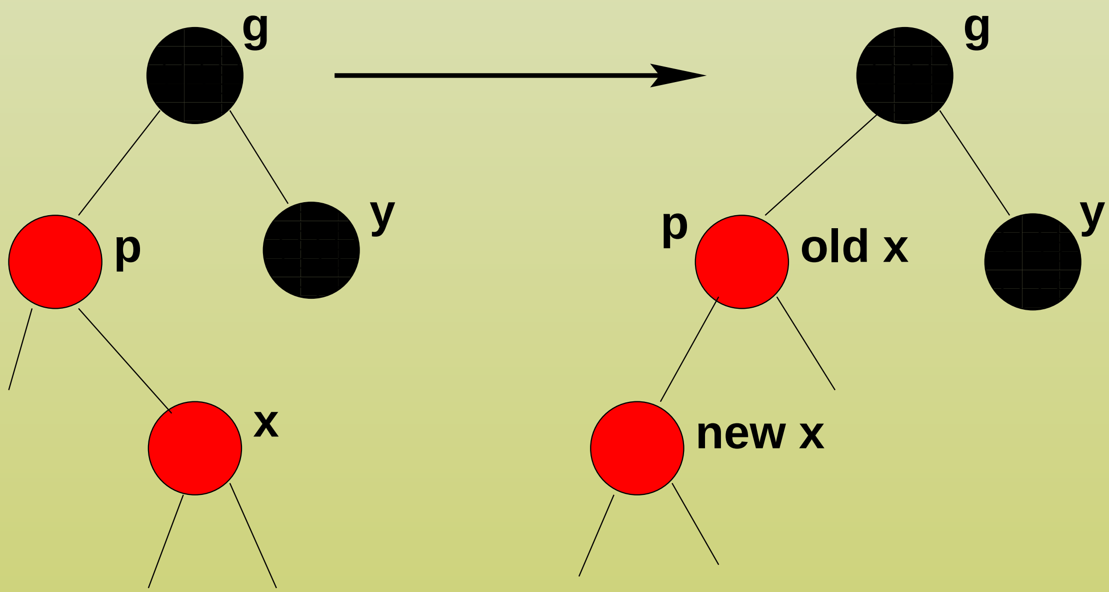
Case 1

At the beginning of each iteration of the `while` loop, `x` points to a red node with a red parent. Case 1 is when `x`'s parent `x->parent` is red and its parent's sibling `x->parent->parent->right` or `x->parent->parent->left` (in the code it is `y`) is also red. The grandparent (`x->parent->parent`) is black. We re-color `x->parent` and `y` black, and the grandparent red. Since the parent of the grandparent may be red, we need to repeat the procedure. Therefore `x` is set to point to the grandparent and we need to repeat the `while` loop with the new node .

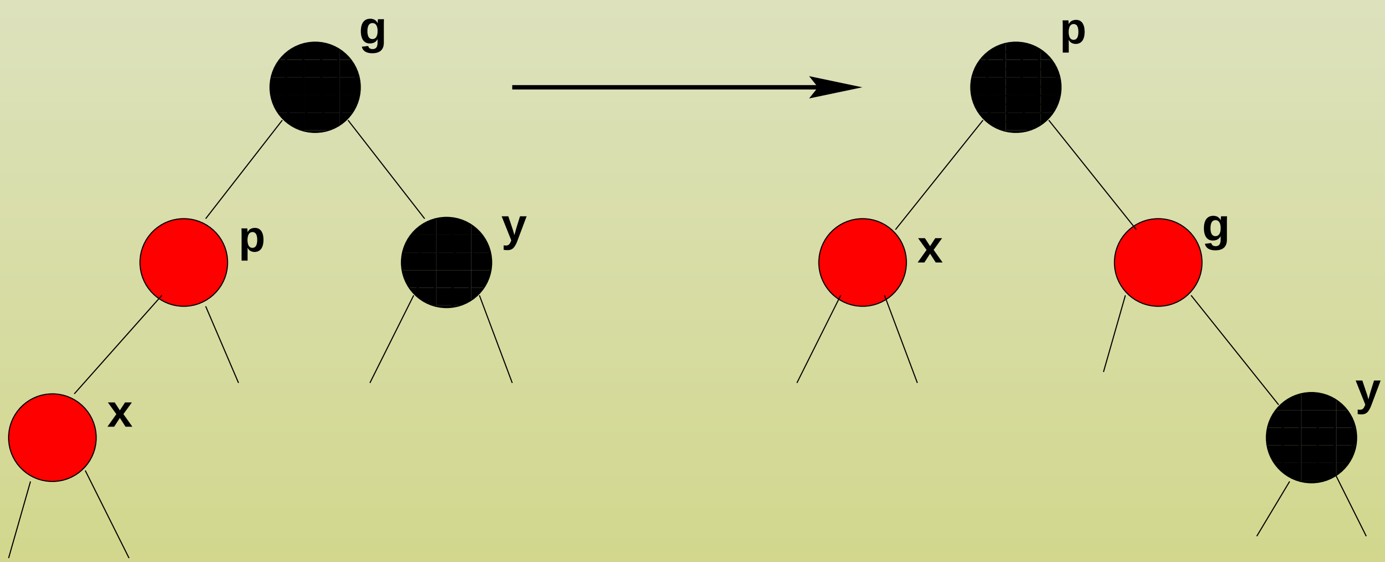


Case 2 and 3

We consider here only the `if` statement (the `else` statement is symmetric). In cases 2 and 3, the color of `y` is black. In case 2 `x` is a right child of `x->parent`, in case 3 it is a left one. In case 2 the left rotation is used to transform it to the case 3. Because `x` and `x->parent` are red, this rotation does not change property 4.



In the case 3 `y` is black. There are color changes for `x`'s parent `x->parent` and grandparent `x->parent->parent`, and then right rotation with respect to `x`'s grandparent. This still preserve property 4. Case 3 does not need any repetition since there are no any two red nodes in a row. The `while` loop is not executed.



Running time in case 1 can be $O(\log_2 n)$ and in cases 2 and 3 it is $O(1)$. The total running time of RBInsert is $O(\log_2 n)$.