Red Black Trees Theory and Analysis

April 3, 2024

Definition

A red-black tree is a binary search tree which has the following red-black properties:

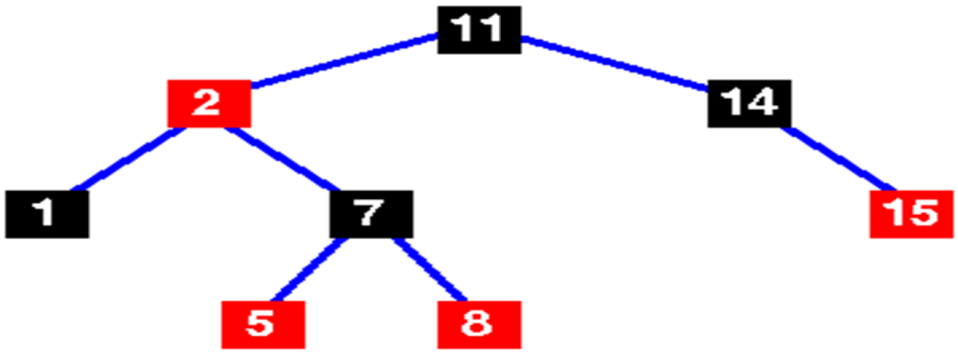
1. Every node is either red or black. By constraining the node colors on any simple path from root to a leaf, RBT's ensure that no such path is twice as long as any other, so that the tree is approximately balanced.
2. Root of the tree must be Black.
3. Every leaf (NULL) is black.
4. If a node is red, then both its children are black. This point implies that on any path from the root to a leaf, red nodes must not be adjacent. However, any number of black nodes may appear in a sequence.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes.

A basic Red Black Tree (Without Sentinels)

A red-black tree is a binary search tree which has the following red-black properties:

1. Every Red Node's children are black. Every leaf node is black (This is on the next slide)
2. Every simple path from a node to a descendant leaf contains the same number of black nodes.

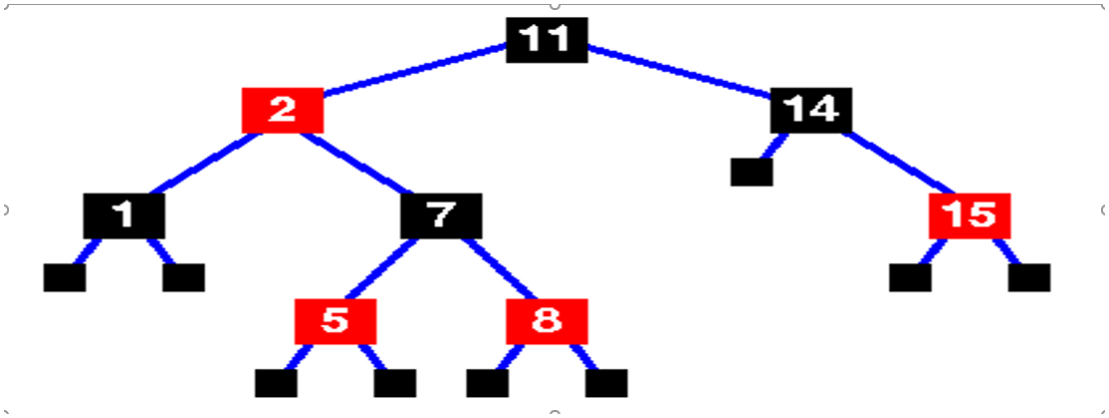
Here is an image illustrating the property:



A basic Red Black Tree (With Sentinels)

Basic red-black tree with the **sentinel** nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

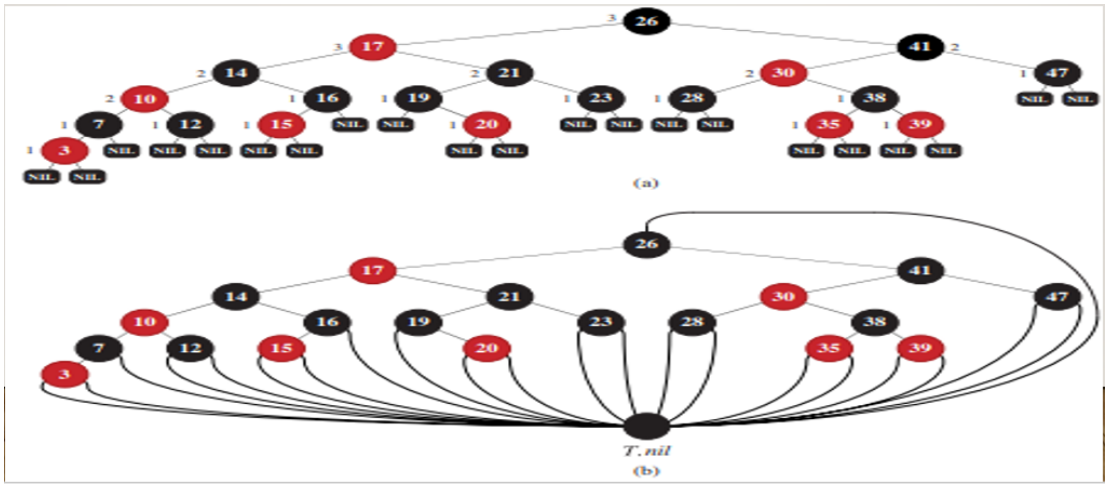
Here is an image illustrating the property:



An alternative design would use a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined. That approach needlessly

wastes space, however, Instead, just the one sentinel **T.nil** represents all the NILs-all leaves and the root's parent.

Here is an image illustrating the property:



Black Height of the Node

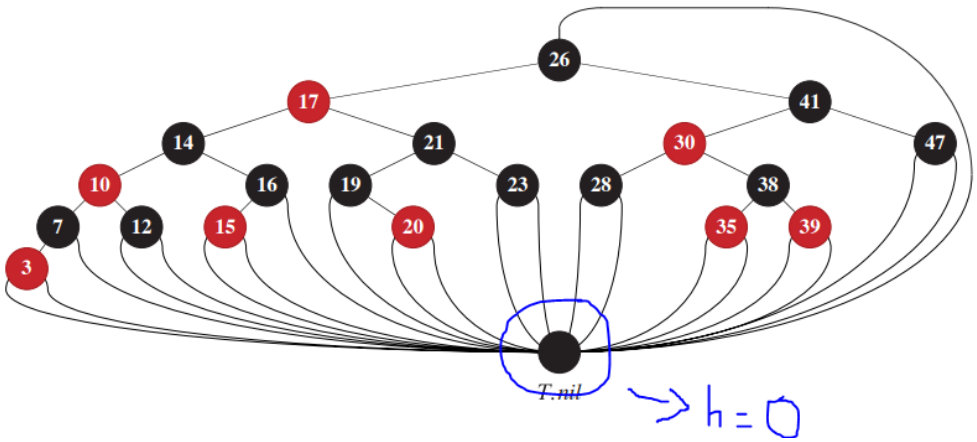
We call the number of black nodes on any simple path from, but not including, a node 'x' down to a leaf, the **black-height** of the node, denoted by **bh(x)**. The black height of a Red Black Tree is the black-height of its root.

Lemma 13.1: A Red Black Tree with 'n' internal nodes has Height at most 2Log(n+1)

Proof:

- Start by showing that the sub-tree rooted at node 'x' contains at least $2^{bh(x)} - 1$ internal nodes.

- Prove by induction. If the height of 'x' is 0, then 'x' must be a leaf node (**T.nil**), and the sub-tree rooted at 'x' contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. (See the figure below)



- Inductive Step:

  Consider a node '**x**' which has a positive height and is an internal node. The node will have two children of which either both or one may be a leaf.

  If a child is black, then it contributes 1 to node x's black height and none to its own. If a child is red, then it contributes none to it's own and node x's black height.

  Therefore, each child has a black-height of either **bh(x)–1** (if child is black) or **bh(x)** (if child is red).

  Since the height of the Child of x is less than node x's height, we apply inductive hypothesis to conclude that each child has at least $2^{bh(x)-1}$–1 internal nodes.

  Thus, the sub-tree at node 'x' contains at least $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1 = 2^{bh(x)}-1$ internal nodes.

- Final Step:

  Let '**h**' be height of the tree. At least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently. the Black Height of the root must be at least **h/2** and so,

  $$\mathbf{n} \geq 2^{h/2} - 1$$

  Rearranging the expression and taking log on both sides results in:

  $$\mathbf{h} \leq 2\log(n+1)$$

Search Runtime

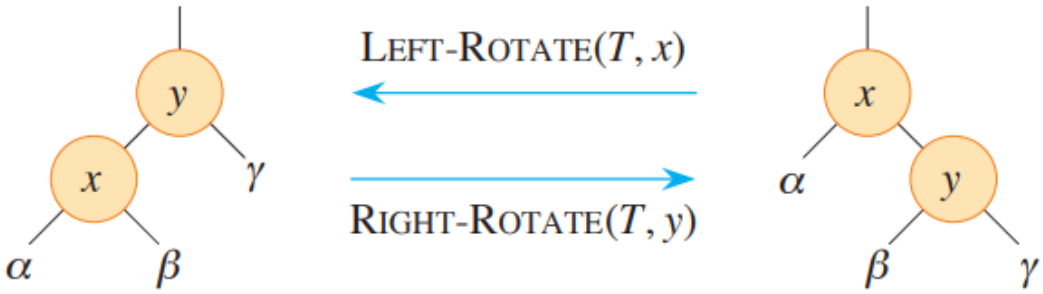
As an immediate consequence of this lemma, the Search operation runs in O(log(n)) time on a Red Black Tree.

Since each run in O(h) time on a binary search tree of height 'h' and any red black tree on 'n' nodes is a binary search tree with height O(log(n))

Rotations

The rotations in a Red Black Tree are similar to what we apply in Binary Search Tree. There are two types of rotations, Left Rotation and Right Rotation. The figure below demonstrates how both Rotations work.

Rotations are performed in O(1) time since only pointers are rearranged to rotate nodes.



Here is an image showing the left and rotate functions of the Red Black Tree:

```cpp
// Function to perform left rotation
void RBTree::rotateLeft(Node *&ptr)
{
    // Store the right child of ptr
    Node *right_child = ptr->right;

    // Update ptr's right child to be the left child of right_child
    ptr->right = right_child->left;

    // Update parent pointers if ptr's right child is not null
    if (ptr->right != nullptr)
        ptr->right->parent = ptr;

    // Update the parent of right_child
    right_child->parent = ptr->parent;

    // Update root if ptr was the root of the tree
    if (ptr->parent == nullptr)
        root = right_child;
    // Update parent's left or right child to be right_child
    else if (ptr == ptr->parent->left)
        ptr->parent->left = right_child;
    else
        ptr->parent->right = right_child;

    // Update the left child of right_child to be ptr
    right_child->left = ptr;

    // Update ptr's parent to be right_child
    ptr->parent = right_child;
}
```

```cpp
// Function to perform right rotation
void RBTree::rotateRight(Node *&ptr)
{
    // Store the left child of ptr
    Node *left_child = ptr->left;

    // Update ptr's left child to be the right child of left_child
    ptr->left = left_child->right;

    // Update parent pointers if ptr's left child is not null
    if (ptr->left != nullptr)
        ptr->left->parent = ptr;

    // Update the parent of left_child
    left_child->parent = ptr->parent;

    // Update root if ptr was the root of the tree
    if (ptr->parent == nullptr)
        root = left_child;
    // Update parent's left or right child to be left_child
    else if (ptr == ptr->parent->left)
        ptr->parent->left = left_child;
    else
        ptr->parent->right = left_child;

    // Update the right child of left_child to be ptr
    left_child->right = ptr;

    // Update ptr's parent to be left_child
    ptr->parent = left_child;
}
```

Insertion

In order to insert a node into a red-black tree with n internal nodes in $\mathbf{O}(\log(n))$ time and maintain the red-black properties, we call a RB-Tree-Insert-Fixup function which performs rotations and colors the nodes to maintain the Red Black Tree property.

We insert nodes in the tree as if it were a Binary Search Tree and color it red. Then, to guarantee that Red Black Tree properties are followed, we call the fixInsertRBTree function which recolors nodes and performs rotations.

Here is an image showing the insert function which inserts a node in the Red Black Tree:

```
// Function to insert a value into the Red Black Tree
void RBTree::insertValue(int n,std::string inp_word)
{
    Node *node = new Node(n,inp_word);
    root = insertBST(root, node);
    fixInsertRBTree(node); // After insertion, fixing the Red Black Tree properties
    std::cout<<"oki insertion done! value: "<<n<<" word: "<<inp_word<<std::endl;
}
```

Here is an image showing the insertBST function which inserts a node following the properties of a BST:

```
// Function to insert a node in the Binary Search Tree (BST)
Node* RBTree::insertBST(Node *&root, Node *&ptr)
{
    if (root == nullptr)
        return ptr;

    if (ptr->data < root->data)
    {
        root->left = insertBST(root->left, ptr);
        root->left->parent = root;
    }

    else if (ptr->data > root->data)
    {
        root->right = insertBST(root->right, ptr);
        root->right->parent = root;
    }

    return root;
}
```

Here is an image showing the fixInsertRBTree function which recolors the nodes and performs rotations (This is only part of the code):

```
// Function to fix Red Black Tree properties after insertion
void RBTree::fixInsertRBTree(Node *&ptr)
{
    // Initialize parent and grandparent nodes
    Node *parent = nullptr;
    Node *grandparent = nullptr;

    // Continue loop until the current node is not the root and both the current node and its parent are red
    while (ptr != root && getColor(ptr) == RED && getColor(ptr->parent) == RED)
    {
        // Update parent and grandparent nodes
        parent = ptr->parent;
        grandparent = parent->parent;

        // Check if parent is the left child of grandparent
        if (parent == grandparent->left)
        {
            // Get the uncle of the current node
            Node *uncle = grandparent->right;

            // Check if uncle is red
            if (getColor(uncle) == RED)
            {
                // Case 1: Uncle is red
                // Recolor parent, uncle, and grandparent
                setColor(uncle, BLACK);
                setColor(parent, BLACK);
                setColor(grandparent, RED);
                // Move up to grandparent
                ptr = grandparent;
            }
            else
            {
```

This image is of the function from the book we have referenced as it provides a summarized logic of this function.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left                  // is z's parent a left child?
 3          y = z.p.p.right                   // y is z's uncle
 4          if y.color == RED                 // are z's parent and uncle both red?
 5              z.p.color = BLACK             }
 6              y.color = BLACK               } case 1
 7              z.p.p.color = RED             }
 8              z = z.p.p                     }
 9          else
10              if z == z.p.right
11                  z = z.p                   }
12                  LEFT-ROTATE(T, z)         } case 2
13              z.p.color = BLACK             }
14              z.p.p.color = RED             } case 3
15              RIGHT-ROTATE(T, z.p.p)        }
16      else // same as lines 3–15, but with "right" and "left" exchanged
17          y = z.p.p.left
18          if y.color == RED
19              z.p.color = BLACK
20              y.color = BLACK
21              z.p.p.color = RED
22              z = z.p.p
23          else
24              if z == z.p.left
25                  z = z.p
26                  RIGHT-ROTATE(T, z)
27              z.p.color = BLACK
28              z.p.p.color = RED
29              LEFT-ROTATE(T, z.p.p)
30  T.root.color = BLACK
```

Working of fixInsertRBTree

The properties violated after the insertion of a node in a Red Black Tree are Property 2 and Property 4.

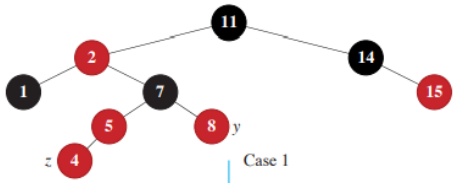
Property 2: Requires the root to be black.

Property 4: Red Node cannot have a Red child.

Property 2 is violated if the inserted node is the root of the tree and Property 4 is violated if inserted node's parent is red.

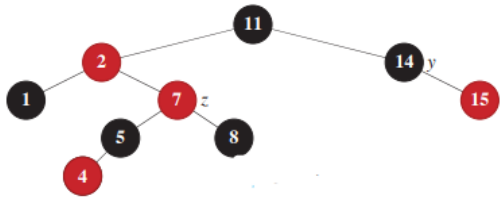
To maintain the properties of a Red Black Tree, there are two cases which occur.

- Case 1: Z's parent is a left child of Z's grandparent.

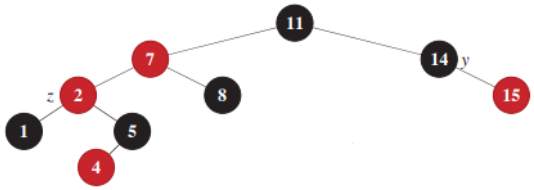


When such a case occurs, there are further two sub-cases which occur,

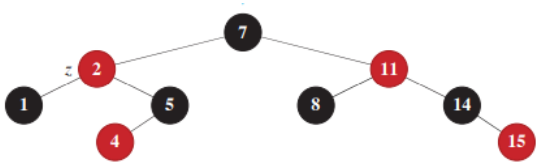
  - When Z's parent and Uncle are both Red. We perform the following operations;
    - Z's Parent color change from RED to BLACK
    - Z's Uncle color change from RED to BLACK
    - Z's Grand-Parent color change from BLACK to RED
    - Make Z it's Grand-Parent.



  - When Z's parent is red but Uncle is Black. We perform the following operations;
    - If Z is right child of Parent then,
      - Make Z its Parent and Left-Rotate over Z.



    - If Z is left child of Parent then,
      - Change Z's Parent color from RED to BLACK.
      - Change Z's Grand-Parent color from Black to RED.
      - Right-Rotate over Z's Grand-Parent



- Case 2: Z's Parent is a right child of Z's Grand-Parent.

  Similar cases occur in this as mentioned above. The only difference is that the Left and Right Rotations are now interchanged.

This way, the Properties of a Red Black Tree will be maintained after a node has been inserted.

Analysis of fixInsertRBTree

Since the height of a Red Black Tree on 'n' nodes is O(log(n)), Case 1 of the function takes O(log(n)) time.

The while loop only repeats if sub-case 1 occurs after which the pointer Z moves two levels up the tree. Hence, the While loop can be executed O(log(n)) times and so the function takes O(log(n)) time to execute.

1 References

Introduction to Algorithms by Thomas H. Cormen:

Introduction to Algorithms PDF