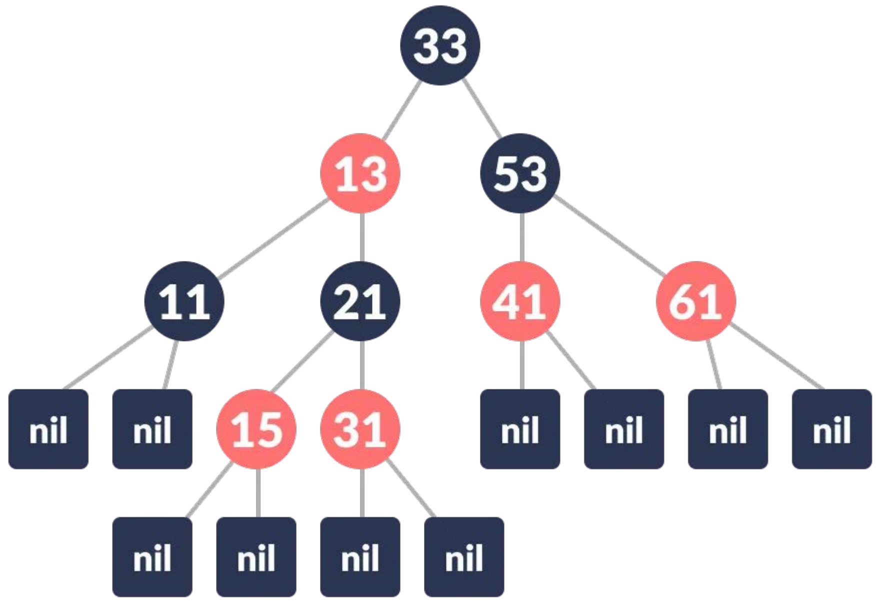
Red Black Trees

Imran Aziz
CS 601
Prof. Derakhshandeh

What is a Red Black Tree?

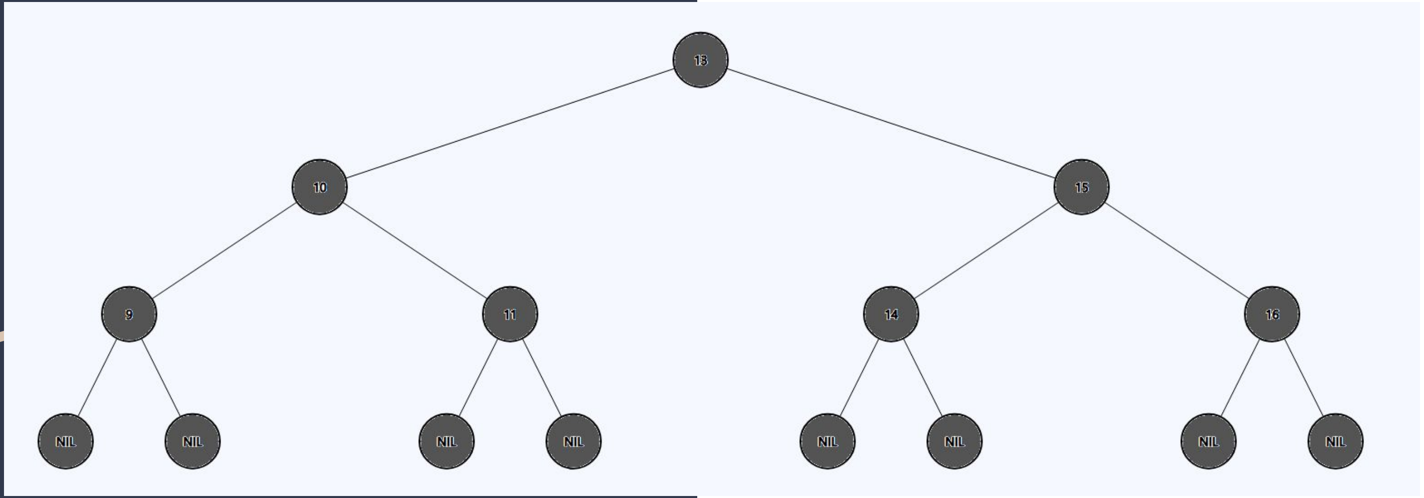
A red black tree is a type of self-balancing BST with the following properties:

1. Every node is either red or black
2. The root node is black
3. Null nodes are black
4. If a node is red, its children are black
5. Every path from a node to each of its null descendants has the same number of black nodes



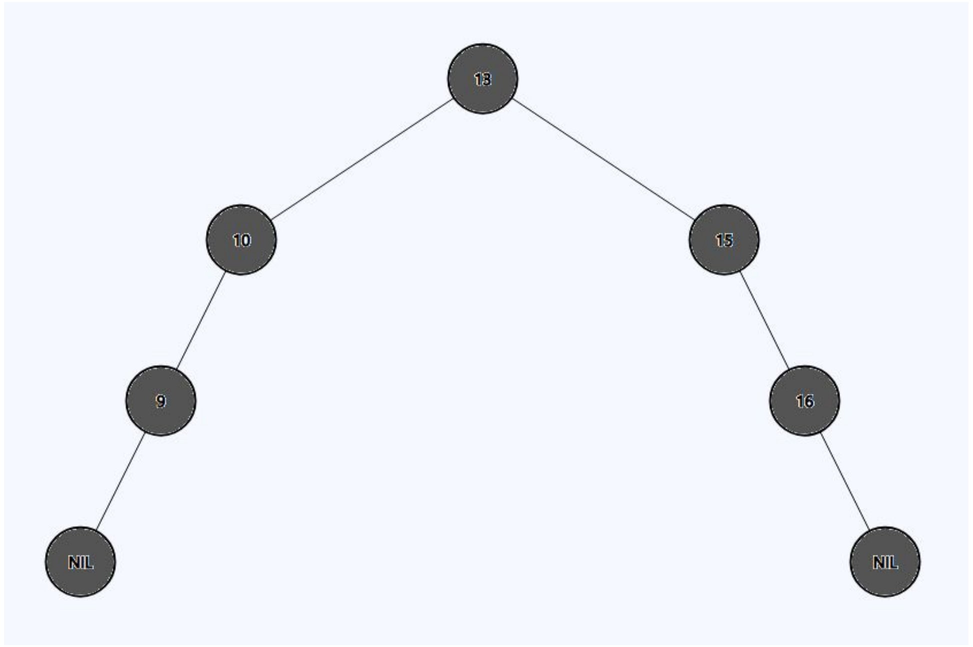
How is a Red Black tree balanced?

Let’s say we have a red black tree that is also a perfect binary tree. The number of nodes from the root to the null nodes will be O(log(n)), as the height of a perfect binary tree is O(log(n)).



How is a Red Black tree balanced?

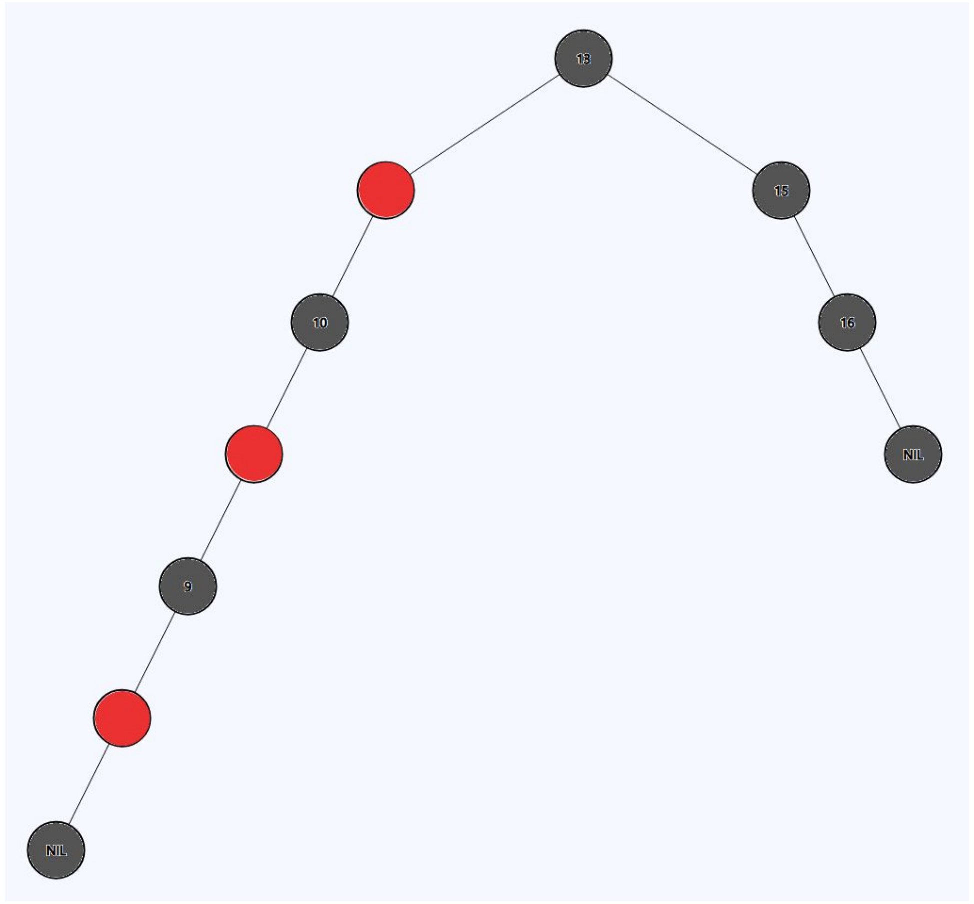
Observe that every path from the root to the null nodes contains two black nodes, and that each path uses the minimum number of nodes required to have two black nodes. We will focus on the leftmost and rightmost paths of the tree.



How is a Red Black tree balanced?

If we extend the left path as far down as possible while maintaining the properties of a red black tree, we end up with the following:

The right path is the shortest valid path with two black nodes and the left path is the longest valid path with two black nodes. The distance between the root and the furthest null node is twice the distance of the root and the nearest null node. Therefore, in the worst case, the height of the tree is O(2*log(n)), which is still O(log(n)).



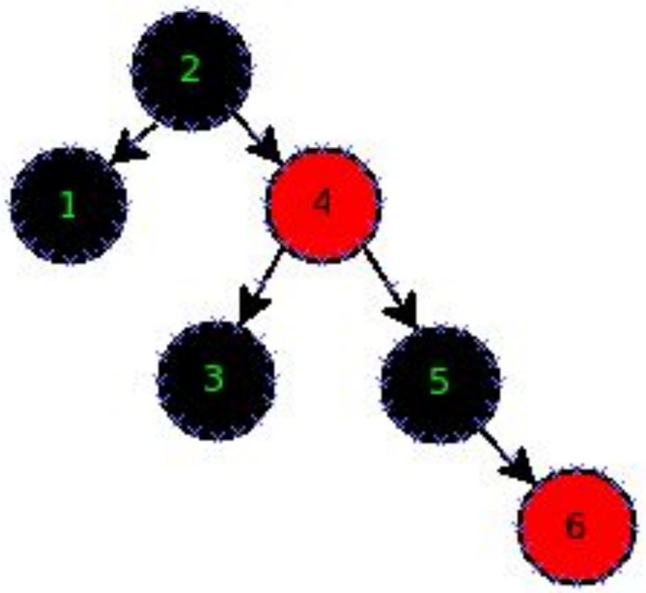
How is a Red Black tree balanced?

In reality, a red black tree is not strictly balanced the same way and AVL tree is, but it is balanced enough to have a height of O(log(n)), making it more efficient in the worst case than a regular binary search tree when it comes to searching, inserting, and deleting.

AVL Tree height: O(1.44log(n))

RB tree height: O(2log(n))

So why do we use red black trees?



Inserting into a Red Black Tree

Properties of a Red Black tree:

1. Every node is either red or black
2. The root node is black
3. Null nodes are black
4. If a node is red, its children are black
5. Every path from a node to each of its null descendants has the same number of black nodes

A node can be inserted into a Red Black tree in the following way:

- Create a new node and color it red, then insert it into the tree the same way you would a normal BST
- If the new node is the root of the tree, color the node black and exit.
- Otherwise, look at the color of the parent node:
  - If it is black, then exit.
  - If it is red, look at uncle's color
    - If uncle is black, rotate, color new "root" as black and old grandparent as red, and exit.
    - If uncle is red, color parent and uncle black
      - If grandparent is root node, exit.
      - Otherwise, color grandparent red and recursively do this process again for grandparent (minus the inserting).

Rotations

Rotations in Red Black trees are the same as the ones in AVL trees.

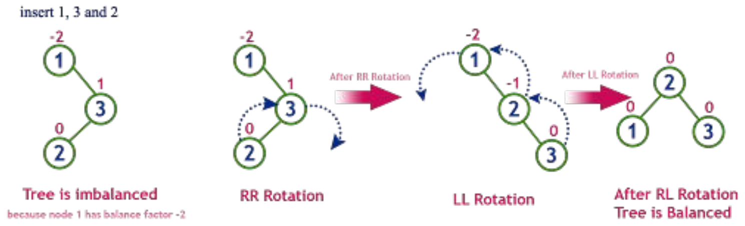
RR rotation - 1 left rotation about grandparent

LL rotation - 1 right rotation about grandparent

RL rotation - 1 right rotation about parent, then 1 left rotation about grandparent

LR rotation - 1 left rotation about parent, then 1 right rotation about grandparent

Why do we need to rotate?

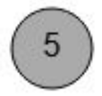


Insertion Examples

Create a RB tree from the following values:

[5,7,6,4]

- Create a new node and color it red, then insert it into the tree the same way you would a normal BST
- If the new node is the root of the tree, color the node black and exit.
- Otherwise, look at the color of the parent node:
    - If it is black, then exit.
    - If it is red, look at uncle's color
        - If uncle is black, rotate, color new "root" as black and old grandparent as red, and exit.
        - If uncle is red, color parent and uncle black
            - If grandparent is root node, exit.
            - Otherwise, color grandparent red and recursively do this process again for grandparent (minus the inserting).

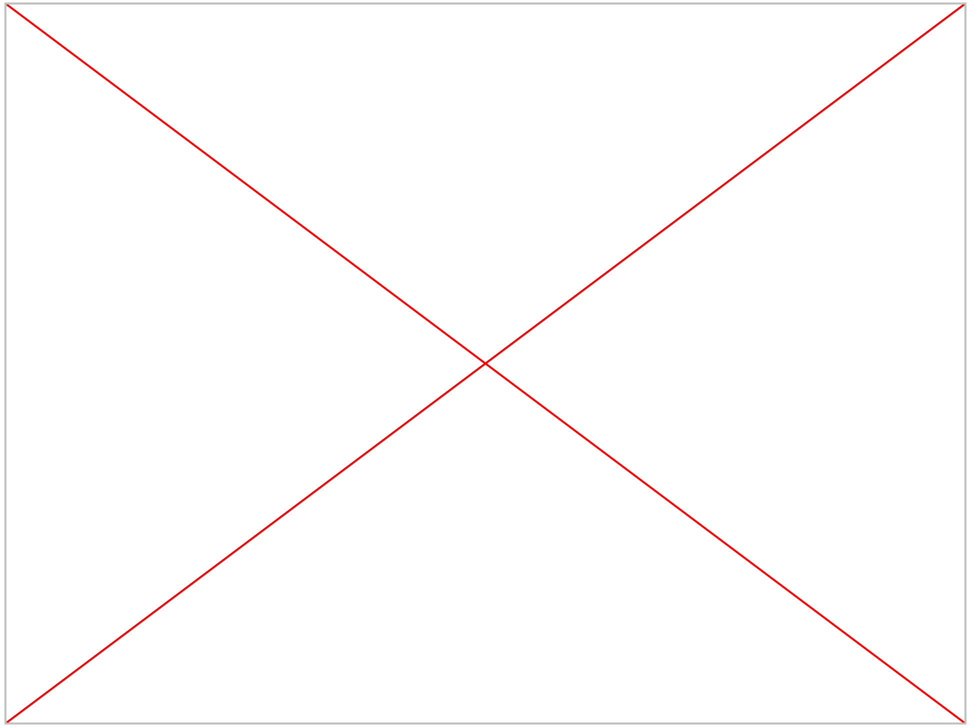


Insertion Examples

Create a RB tree from the following values:

[5,7,6,4]

- Create a new node and color it red, then insert it into the tree the same way you would a normal BST
- If the new node is the root of the tree, color the node black and exit.
- Otherwise, look at the color of the parent node:
  - If it is black, then exit.
  - If it is red, look at uncle's color
    - If uncle is black, rotate, color new "root" as black and old grandparent as red, and exit.
    - If uncle is red, color parent and uncle black
      - If grandparent is root node, exit.
      - Otherwise, color grandparent red and recursively do this process again for grandparent (minus the inserting).

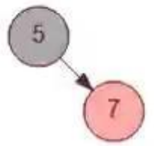


Insertion Examples

Create a RB tree from the following values:

[5,7,6,4]

- Create a new node and color it red, then insert it into the tree the same way you would a normal BST
- If the new node is the root of the tree, color the node black and exit.
- Otherwise, look at the color of the parent node:
  - If it is black, then exit.
  - If it is red, look at uncle's color
    - If uncle is black, rotate, color new "root" as black and old grandparent as red, and exit.
    - If uncle is red, color parent and uncle black
      - If grandparent is root node, exit.
      - Otherwise, color grandparent red and recursively do this process again for grandparent (minus the inserting).

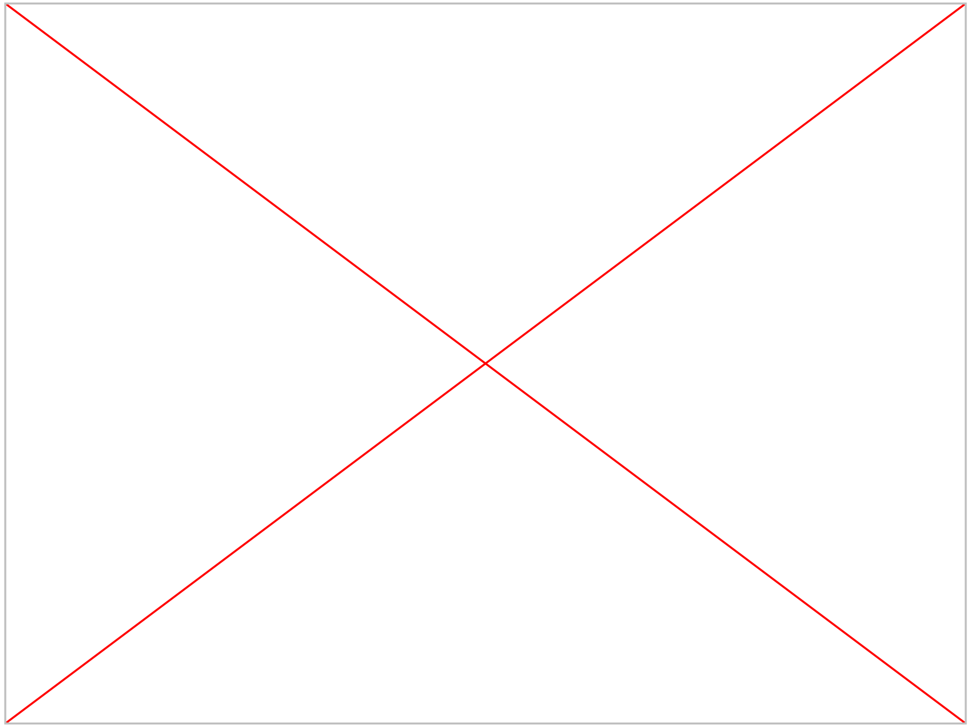


Insertion Examples

Create a RB tree from the following values:

[5,7,6,4]

- Create a new node and color it red, then insert it into the tree the same way you would a normal BST
- If the new node is the root of the tree, color the node black and exit.
- Otherwise, look at the color of the parent node:
    - If it is black, then exit.
    - If it is red, look at uncle's color
        - If uncle is black, rotate, color new "root" as black and old grandparent as red, and exit.
        - If uncle is red, color parent and uncle black
            - If grandparent is root node, exit.
            - Otherwise, color grandparent red and recursively do this process again for grandparent (minus the inserting).



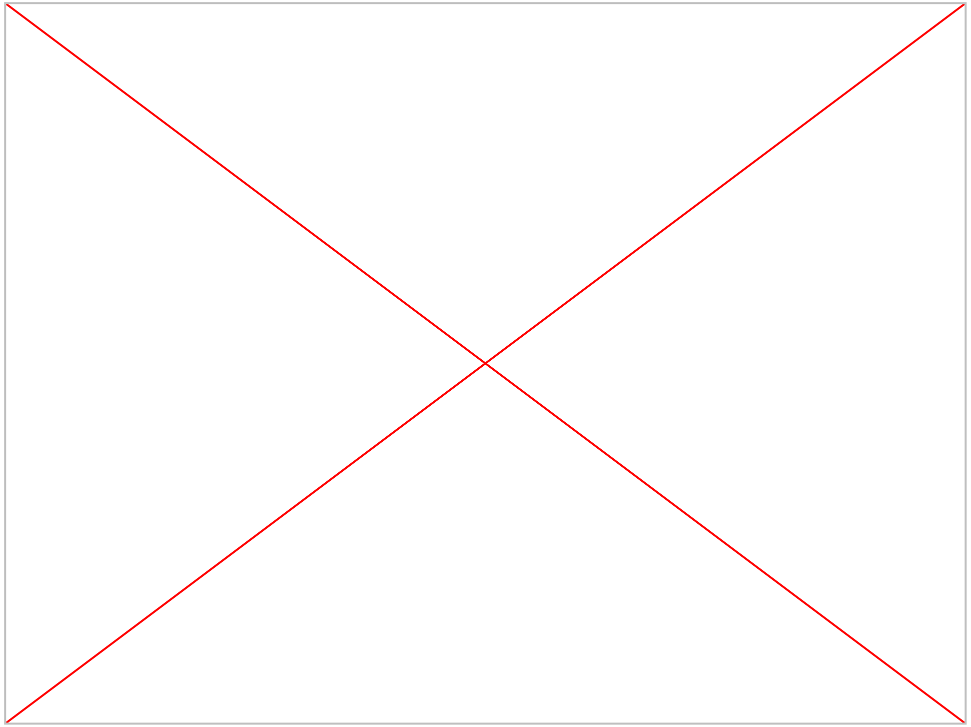
Deleting from a Red Black Tree

- Find node to delete, take note of color before removing.
- If it was red, exit.
- If it was black, take some extra steps:
  - Look at new node that took its place, we will call it x. (If the node had no children, x will be a null node)
  - While x is not the root and x is black, do the following:
    - If x’s sibling is red:
      - color the sibling black, color the parent red, and rotate the parent towards x
    - If x’s sibling is black and has 2 black children
      - color the sibling red, and x becomes its parent
    - If x’s sibling is black, its furthest child from x is black, and it's closer child is red
      - color the closer child black, color the sibling red, then rotate the sibling away from x
    - If x’s sibling is black, its furthest child from x is red, and it's closer child is black
      - the sibling takes its parent color, the parent becomes black, the sibling’s further child becomes black. Rotate x’s parent towards x, and x becomes the root of the tree
  - Once the loop is over, color x black

Deletion Examples

- Find node to delete, take note of color before removing.
- If it was red, exit.
- If it was black, take some extra steps:
    - Look at new node that took its place, we will call it x. (If the node had no children, x will be a null node)
    - While x is not the root and x is black, do the following:
        - If x’s sibling is red:
            - color the sibling black, color the parent red, and rotate the parent towards x
        - If x’s sibling is black and has 2 black children
            - color the sibling red, and x becomes its parent
        - If x’s sibling is black, its furthest child from x is black, and it's closer child is red
            - color the closer child black, color the sibling red, then rotate the sibling away from x
        - If x’s sibling is black, its furthest child from x is red, and it's closer child is black
            - the sibling takes its parent color, the parent becomes black, the sibling’s further child becomes black. Rotate x’s parent towards x, and x becomes the root of the tree
    - Once the loop is over, color x black

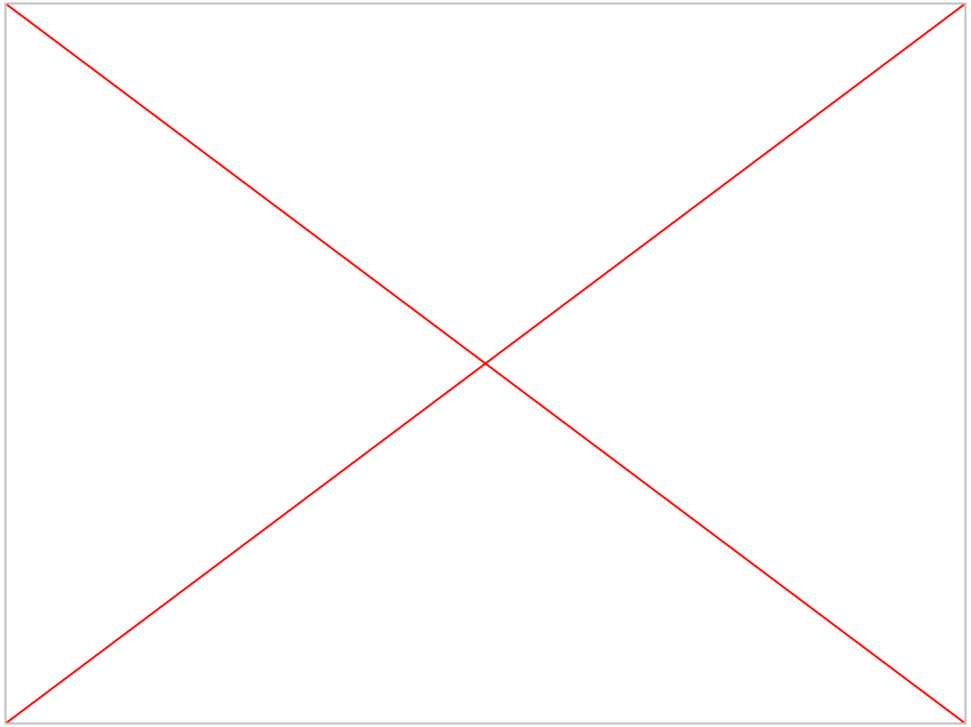
Deleting 4



Deletion Examples

- Find node to delete, take note of color before removing.
- If it was red, exit.
- If it was black, take some extra steps:
    - Look at new node that took its place, we will call it x. (If the node had no children, x will be a null node)
    - While x is not the root and x is black, do the following:
        - If x’s sibling is red:
            - color the sibling black, color the parent red, and rotate the parent towards x
        - If x’s sibling is black and has 2 black children
            - color the sibling red, and x becomes its parent
        - If x’s sibling is black, its furthest child from x is black, and it's closer child is red
            - color the closer child black, color the sibling red, then rotate the sibling away from x
        - If x’s sibling is black, its furthest child from x is red, and it's closer child is black
            - the sibling takes its parent color, the parent becomes black, the sibling’s further child becomes black. Rotate x’s parent towards x, and x becomes the root of the tree
    - Once the loop is over, color x black

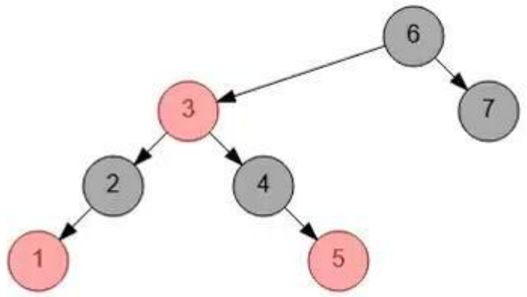
Deleting 5



Deletion Examples

- Find node to delete, take note of color before removing.
- If it was red, exit.
- If it was black, take some extra steps:
    - Look at new node that took its place, we will call it x. (If the node had no children, x will be a null node)
    - While x is not the root and x is black, do the following:
        - If x’s sibling is red:
            - color the sibling black, color the parent red, and rotate the parent towards x
        - If x’s sibling is black and has 2 black children
            - color the sibling red, and x becomes its parent
        - If x’s sibling is black, its furthest child from x is black, and it's closer child is red
            - color the closer child black, color the sibling red, then rotate the sibling away from x
        - If x’s sibling is black, its furthest child from x is red, and it's closer child is black
            - the sibling takes its parent color, the parent becomes black, the sibling’s further child becomes black. Rotate x’s parent towards x, and x becomes the root of the tree
    - Once the loop is over, color x black

Deleting 7



RB Tree vs AVL Tree

AVL trees and RB trees are both technically O(log(n)) for searching, inserting, and deleting, but an AVL tree has a height of O(1.44*log(n)), while a RB tree has a height of O(2*log(n)). This means that an AVL tree is more efficient for searching in the worst case.

However, because a RB tree is only roughly balanced, there will be less rotations that will need to occur on average when inserting into or deleting from a RB tree. This makes inserting and deleting more efficient for RB trees.

So, we can use a RB tree when we are going to be doing more insertions and deletions, and an AVL tree when doing more searches.

	AVL Tree	RB Tree
Searching	O(log(n))	O(log(n))
Inserting	O(log(n))	O(log(n))
Deletion	O(log(n))	O(log(n))

Real World Uses

C++ STL: map, multimap, multiset

“To quote Linux Weekly News:

> There are a number of red-black trees in use in the kernel.
> The deadline and CFQ I/O schedulers employ rbtrees to
> track requests; the packet CD/DVD driver does the same.
> The high-resolution timer code uses an rbtree to organize outstanding
> timer requests. The ext3 filesystem tracks directory entries in a
> red-black tree. Virtual memory areas (VMAs) are tracked with red-black
> trees, as are epoll file descriptors, cryptographic keys, and network
> packets in the ‘hierarchical token bucket’ scheduler.”

https://www.kernel.org/doc/Documentation/rbtree.txt

Thank you