A binary search tree is a data structure that allows us to easily search for elements. A binary search tree is created by inserting elements to the left of every node that it is smaller than and to the right of every node that it is larger than. So when searching for an element we simply check if each node is larger or smaller than the element being search for and move to the right or left node accordingly.

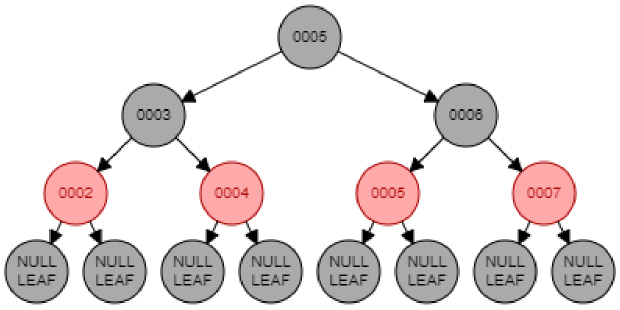
In many cases a binary search tree is quite efficient but there are some very inefficient cases. For example, if the elements being added to the tree are strictly decreasing we have a tree that has a height equal to the number of elements being added with each node in the tree only containing left children. The way to avoid heavily skewed trees is to create a balanced binary search tree. A tree is balanced if at every node the left subtree and right subtree are not much different in height. It is possible to change your definition of “much different” but a common definition is a tree is balanced if each node’s subtree does not differ in heigh by more than 1. This creates a tree that has elements evenly distributed on the right and left half of the tree. So the height of the total tree will be much lower than the heavily skewed tree mentioned earlier which reduces search times for the lower element in the tree.

A red black tree is a self balancing binary search tree. Meaning when inserting and deleting elements the tree maintains its balance. Each node has an extra bit that stores whether the node is red or black. A red black tree must meat the following properties

1. The root property: The root node is black.
2. The external property: All leaves on the tree are black.
3. The red property: All red nodes have black children
4. Depth property: All external nodes have a path from the root going through the same number of black nodes

By making sure these properties are maintained after insertion and deletion the tree balances itself.

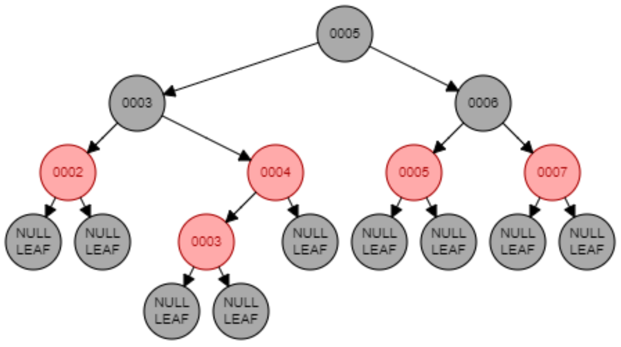
Here is an example of a red black tree.



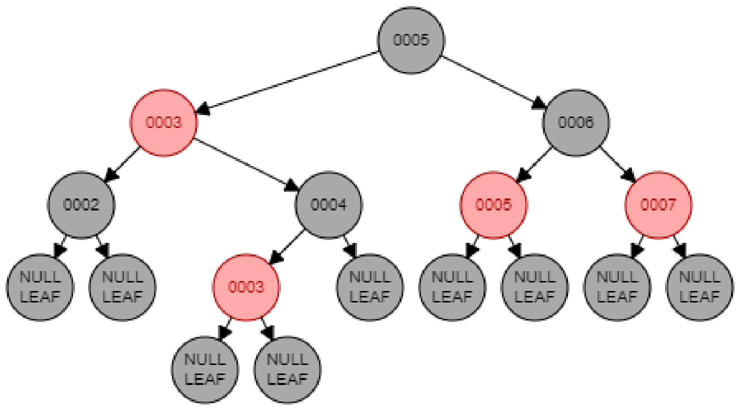
Notice what happens when a node is inserted.

Firstly, each time a node is added it is first added as a red node. Let’s add a 3 to the tree

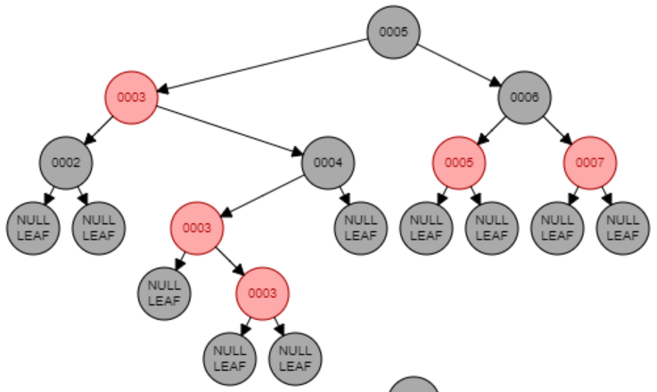
The 3 being added now violates the red property because we have a red node as the child of a red node. So we have to rearrange the tree a little bit. The 3 could be changed to black to fix the red property but then its children would violate the depth property.



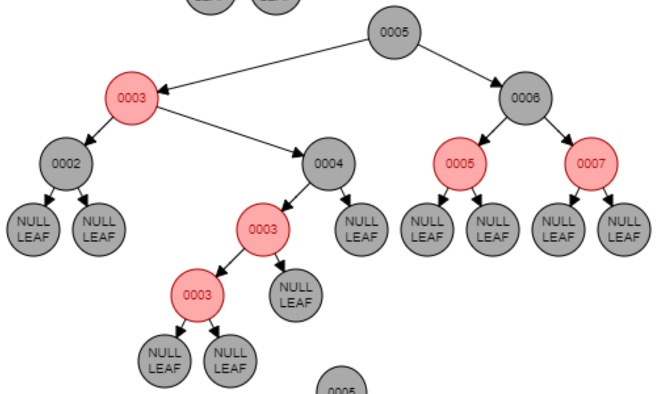
Instead, we change the 4 to black. In order to maintain the black depth property we then change the 2 that is a sibling of 4 to black, and the 3 that is a parent of 4 to red. Now we have a balanced search tree that maintains the red black tree properties.



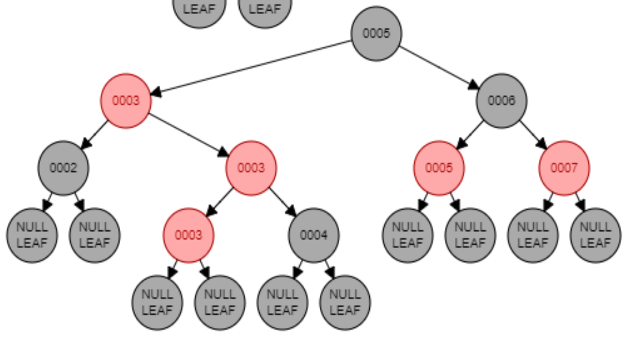
Now if we add a 3 again we have the same problem where we have a red node with a red child. But we cannot just swap colors because it will always fail the depth property due to the balance of the tree being off



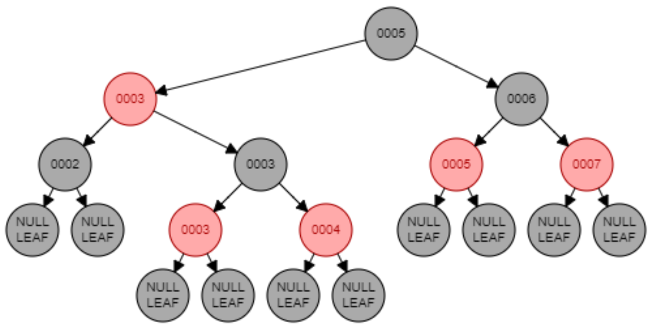
So first we move 3 to be the left child instead of the right child because its parent is a left child.



Then we rotate to the right which balances the tree



Then we fix the colors.



So now the red black tree matches all of the properties. And as you can see it rebalanced itself in the process.

Here is a pseudocode explanation of the algorithm to insert an element into a red black tree

First we insert a red node by traversing through the tree going left when the element is smaller than the node and right when the element is larger than the node. If the tree is empty we just make the new node the root and color it black. Once we reach the external nodes we simply add the new node to the tree. Then we need to rebalance the tree which is done in the rebalanceInsert function

```
def insert(self, key):
    initialize a new node of key
    initialize the parent of node to None
    initialize the left child to None
    initialize the right child to None
    set the color of Node to red

    parse through the tree:                                          O(logn)
        at each element check if it is larger than node:
            if the element is large than node go to the right otherwise go to the left


    set the parent of node to be the last element found above

    if the parent of node is none:
        make node the root of the tree and color it black
    else if node is less than its parent:
        make node the left child of its parent
    else:
        make node the right child of its parent

    call rebalanceInsert(node)                               `O(?)
```

Inserting a new node to the tree will traverse down the tree. So it will be O(tree height). Since the tree is always balanced the height will be logn. So insertion is O(logn) before we take into account the rebalanceInsert function.

Rebalancing the tree is what maintains the red black properties after insertion.
The first thing we check is whether or not the parent of newNode is red. If the parent is black then the red black properties are not violated and the tree does not need to be rebalanced.
If the parent of newNode is red then we check to see if the uncle(sibling of the parent) of newNode is red. If it is black we can replace the grandparent with the parent and make the grandparent a sibling of newNode. We then need to color the parent black and the grandparent

red. Since the uncle was black changing the granparent to red does not cause a failure of the red property. If the uncle is red it would cause a failure so we need to recolor the nodes before moving them around. The actual implementation of this is more complicated because it has to take into account whether or not the nodes are left or right children in order to know which way to rotate them. But it is still doing what is described above.

```
def rebalanceInsert(self, newNode):
    set p to be the parent of newNode
    set gp to be the grandparent of newNode
    while p is red:                                          O(logn)
        if p is the left child of gp:
            set z to be the right child of gp
            if z is red:
                set z to black
                set p to black
                set gp to red
                set newNode to gp
            else:
                if newNode is the right child of p:
                    set newNode to p
                    left rotate newNode
                set p to black
                set gp to red
                right rotate gp
        else:
            set z to the left child of gp

            if z is red':
                set z to black
                set p to black
                set gp to red
                set newNode to gp
            else:
                if newNode is the left child of p:
                    set newNode to p
                    left rotate newNode
                set p to black
                set gp to red
                left rotate gp
        if newNode is the root node:
            break
    set the root node to black
```

The function to rotate a node to the left takes a node x with parent p and right child y. It shifts the tree so x is not the left child of y and y is now the child of p. Then the left child of y is now the left child of x. The right rotate does the same thing in the other direction. It takes a node x with left child y and parent p and shifts the tree so x is the right child of y and y is the child of p.

```
def rotateL(self, x):
    initialize y to be the right child of x
    set the right child of x to be the left child of y
    if the left child of y is not null:
        set the parent of the left child of y to be x

    set the parent of y to be the parent of x
    if x does not have a parent:
        set the root of the tree to be y
    elif x is the left childt:
        set the left child of the parent of x to be y
    else:
        set the right child of the parent ofo x to be y
    set the left child of y to be x
    set the parent of x to be y

  def rotateR(self, x):
    do the same thing as left rotate but swap every left and right
```

The rotate functions have no loops and every line runs in constant time so they are O(1).

In the worst case of rebalancing we have to continuously change colors for every element following a path from the new node to the root. In that case we are recoloring O(logn) times because the height of the tree is logn. Since the rest of the operations, including rotate, are O(1) the overall run time of rebalance is O(logn)

So the total insert function is O(logn + logn) = O(logn)

Deleting a node is a little bit more complicated than inserting one. There are a few cases that need to be considered.
If the node has one child we can delete the node and put its child where the node used to be. We then color that node black to preserve the depth property
If the node has no children we simply delete that node and put an external node in its place
If the node has two children we swap the node with the minimum of the right subtree and then delete it. That way it can easily be deleted since it won’t have any children

If we delete a red node the red black tree properties will be preserved. If we delete a black node we have to do some restructuring because the depth property is no longer met.

```
def deleteNode(self, key):
    initialize node to the root of the tree
    initialize the variable toBeDeleted to null and call it tbd
    while node is not null:                                          O(logn)
        if node equals key:
            set tbd to node

        if node is less than key:
            move to the right child of node
        else:
            move to the left child of node

    if tbd is null:
        print("Key not Found")
        return

    initialize original_color to the color of tbd and call it oColor
    if the left child of tbd is null:
        initialize x to the right child of tbd
        call transplant of tbd and x
    elif the right child of tbd is null:
        initialize x to the left child of tbd
        call transplant of tbd and x
    else:
        initialize y to the minimum of the right subtree of tbd      O(?)
        set oColor to the color of y
        initialize x to the right child of y
        if y is a child of tbd:
            set the parent of x to y
        else:
            call transplant of y and the right child of y
            set the right child of y to the right child of tbd
            set y to be the parent of the right child of y

        call transplant of tbd and y
        set the left child of y to the left child of tbd
        set the parent of the left child of y to be y
        set the color of y to be oColor
    if oColor is black:
        call the rebalanceDelete function                            O(?)
```

The most time consuming parts of the delete function are the search for the element to be deleted, the min function, and the rebalanceDelete function. Searching the tree is O(logn) because it depends on the heigh of the tree. The other two functions are analyzed below.

```
def min(self, node):
    while the left child of node is not null:                    O(logn)
        move to the left child of node
    return node
```

The function to find the minimum value in the tree will traverse through the height of the tree so it is O(logn)

```
def transplant(self, toRemove, toReplace):
    if toRemove has no parent:
        set the root of the tree to toReplace
    elif toRemove is a left child:
        set the left child of the parent of toRemove to be toReplace
    else:
        set the right child of the parent of toRemove to be toReplace
    set the parent of toReplace to be the parent of toRemove
```

Transplant replaces one node with another node. There are no loops and every line runs in constant time so transplant is O(1)

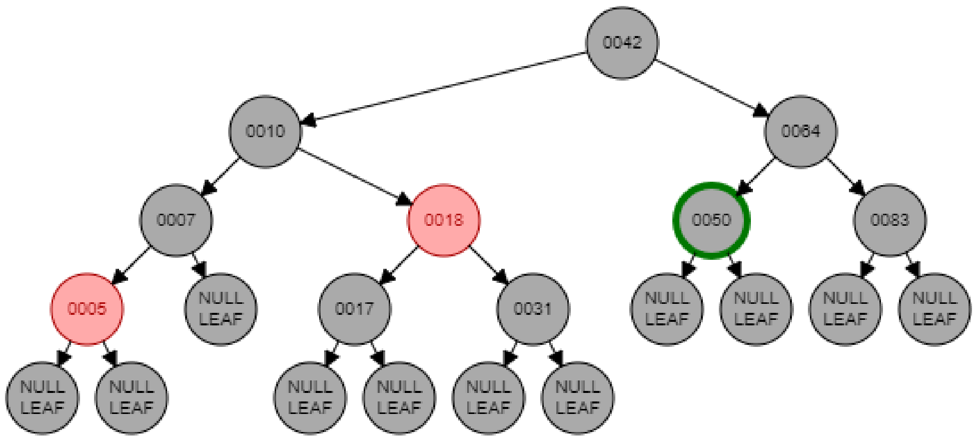
When deleting a black node rebalanceDelete is called to fix the depth property. The cases are based on the sibling of the node that is violating the depth property.

```
If the node’s sibling is red:
        Set the sibling to black
        set the parent of node to red
        rotate the parent of node to the left (right if node is a right child)
Next we look at the children of the sibling of node.
If they are both black:
        set the sibling to red and set x to its parent
If only one is black:
        set the other child to black
        set the sibling of node to red
        rotate the sibling of node
If neither is black:
        set the sibling of node to the same color as its parent
        set the color of the parent to black
        set the right child of the sibling to be black (left if node is a right child)
        rotate the parent of node to the left (right if node is a right child)
        set the root of the tree to the node
```

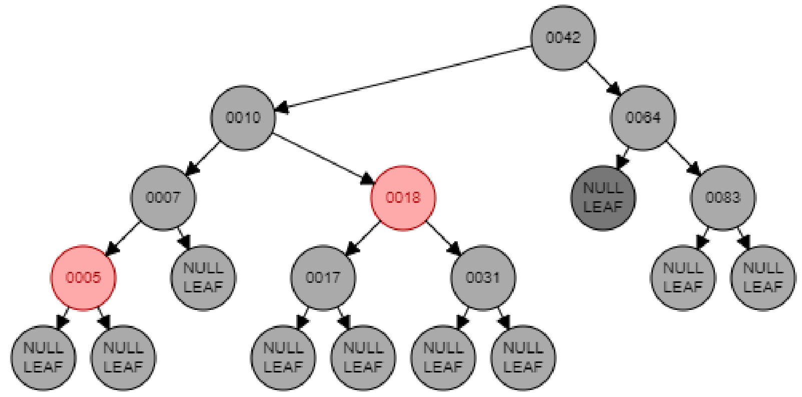
These steps are repeated until node becomes the root. Because once node is the root it means the depth property is satisfied.

```
def rebalanceDelete(self, x):
    while x is not the root and x is not black:                              O(logn)
        if x is the left child of its parent:
            initialize w to the right child of the parent of x

            if w is red:
                set w to black
                set the parent of x to red
                left rotate the parent of x
                set w to the right child of the parent of x

            if the left and right child of w are both black:
                set w to black
                set x to be the parent of x
            else:
                if the right child of w is black:
                    set the left child of w to black
                    set w to red
                    right rotate w
                    set w to be the right child of the parent of x

                set the color of w to the color of the parent of x
                set the parent of x to black
                set the right child of w to black
                left rotate the parent of x
                set x to the root of the tree
        #same as above with right and left swapped
        else:
            repeat all the steps above but switch every left and right
    set x to black
```

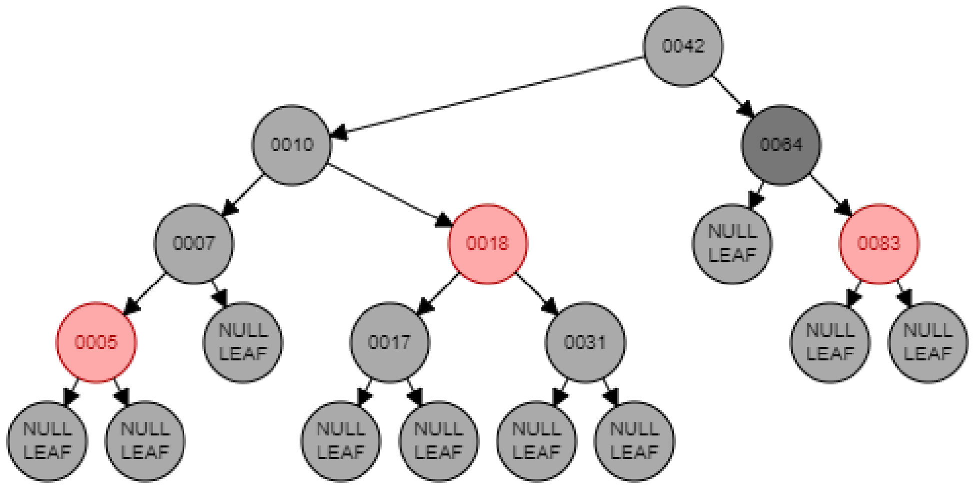
Lets look at an example of removing a node. Here we will removed the node 50 marked with the green circle.



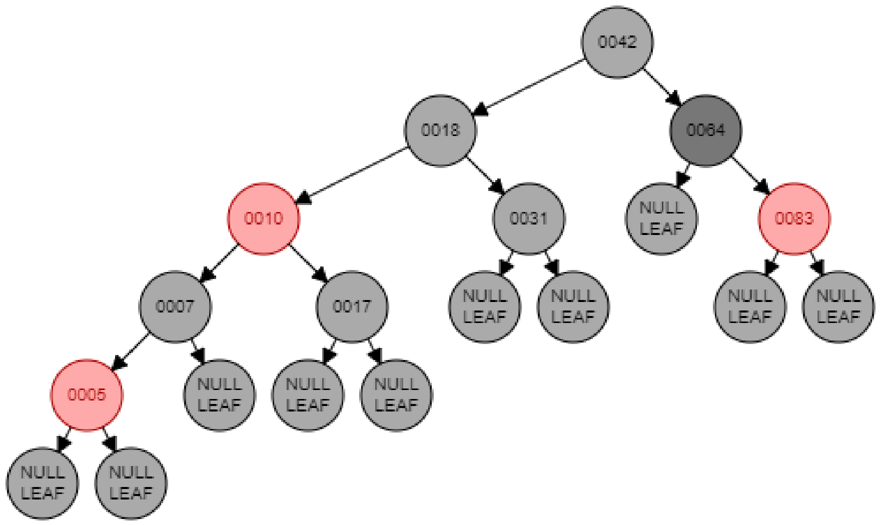
The node does not have any children so it is removed and one of the external nodes is moved up to take its place. In order to preserve the black depth the external node is marked double black.



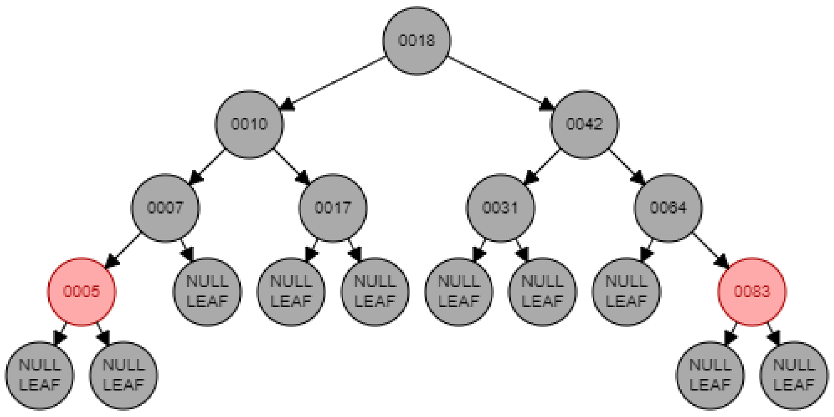
Now we start the restructuring to maintain the red black properties. Since the sibling of the node(83) in question is black with two black children it is changed to red. The parent node is then marked double black



Now we move to the node 64. Its sibling is black with 1 black child so 18 is changed to black and 10 is changed to red. Then the sibling is rotated to the left so 18 is now the child of 42.



Finally, the sibling of 64 is black so 10 is colored black, 18 is colored red, and 18 is shifted right. Then 18 is the root so it is colored black



Now we have a balanced tree that fits every property of a red black tree.

This solution differs very slightly from the implementation I am using. In my code the 83 node would be black in the third step, and another step would be added to turn 83 red. But the algorithm is essentially doing the same thing and it reaches the same result.

Here is the deletion of 50 using the implementation I am using.
Deleting 50

1.
```
R----42(BLACK)
   L----10(BLACK)
   |  L----7(BLACK)
   |  |  L----5(RED)
   |  R----18(RED)
   |     L----17(BLACK)
   |     R----31(BLACK)
   R----64(BLACK)
      R----83(BLACK)
```

2.
```
R----42(BLACK)
   L----10(BLACK)
   |  L----7(BLACK)
   |  |  L----5(RED)
   |  R----18(RED)
   |     L----17(BLACK)
   |     R----31(BLACK)
   R----64(BLACK)
      R----83(BLACK)
```

3.
```
R----42(BLACK)
   L----10(BLACK)
   |  L----7(BLACK)
   |  |  L----5(RED)
   |  R----18(RED)
   |     L----17(BLACK)
   |     R----31(BLACK)
   R----64(BLACK)
      R----83(BLACK)
```

4.
```
R----42(BLACK)
   L----10(BLACK)
   |  L----7(BLACK)
   |  |  L----5(RED)
   |  R----18(RED)
   |     L----17(BLACK)
   |     R----31(BLACK)
   R----64(BLACK)
      R----83(RED)
```

5.
```
R----42(BLACK)
   L----18(BLACK)
   |  L----10(RED)
   |  |  L----7(BLACK)
   |  |  |  L----5(RED)
   |  |  R----17(BLACK)
   |  R----31(BLACK)
   R----64(BLACK)
      R----83(RED)
```

6.
```
R----18(BLACK)
   L----10(BLACK)
   |  L----7(BLACK)
   |  |  L----5(RED)
   |  R----17(BLACK)
   R----42(BLACK)
      L----31(BLACK)
      R----64(BLACK)
         R----83(RED)
```

The rebalanceDelete function, in the worst case, will repeat for every parent of the node until you reach the root. That means the number of times that the while loop repeats depends on the heigh of the tree. So it is O(logn). Inside the while loop there are no loops and every line is O(1). So the rebalanceDeletefunction is O(logn)

So the overall delete function is the run time of search + the run time of min + the run time of rebalanceDelete.
So deleting a node from a red black tree is O(logn) + O(logn) + O(logn) = O(logn)

Print in order recursively parses through the tree first moving left to find the smallest elements. Then printing those elements, then moving right to reach the remaining elements.

```
def print_in_order(self, node):                                        O(n)
    if the left child of node is not null:
        call print_in_order on the left child of node
    print node
    if if the right child of node is not null:
        call print_in_order on the right child of node
```

The print_in_order function will reach every element in the red black tree exactly 1 time and print it. So the function will always have O(n) performance.

In conclusion a red black tree seems to be a very effective data structure. We have O(logn) insertion, deletion, and searching. While there are data structures that can do some of these operations faster a red black tree shows consistency across all operations. Many data structures can improve on one operations speed while reducing the speed of another. Compared to a normal binary search tree, a heap, a queue, or many of the other data structures we have studied thus far a red black tree is more complicated and was more difficult to implement. But that is a very small price to pay for the increase in performance.

Citations:
My implementation is based heavily off an implementation found here
https://www.programiz.com/dsa/red-black-tree
I wrote my code basing it off of the pseudocode explanations of the algorithms. There was also a full implementation on the website and I referenced the code when I got stuck. I did have to use the transplant and rotate functions from the website. Additionally I used one of their functions to print the tree in my deletion example. I think after writing the report I understand the implementation much better and if I were to implement it again I would need less assistance.

For the red black tree immages I got them from this website:
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html