Comparing Different Self-Balancing Trees

Fremont, Lucy, Annika

The Questions:

Which self-balancing tree is most time efficient for searching and inserting?

Aka which self-balancing tree is more efficient.

Which self-balancing tree was most difficult to implement?

AVL Tree

-Binary search tree

-Rebalances are based on heights-two bits

-Balance Factor: height difference between of its two child subtrees

-Rebalances are rotations: Right-Right, Right-Left, Left-Left, Left-Right

-More strictly balanced

-We have looked at AVL trees so will skip visualization

-Unfortunately did not get a working version

Red-Black Trees

- Binary search trees
- Rebalances are based on color-one bit
  - black depth-height balanced where each path from root to leaf must have same number of black nodes.
  - red node cannot have a red node child
  - Root is always black
- rebalances are recoloring and rotations:

Case 1: If current node is not root and parent node and pibling node are red-recolor and repeat steps with grandparent node as current node

Case 2: if current node is not root and parent node is red and pibling is black-rotate

- Rotations are same as AVL
- Not as strictly balanced than AVL

More Discussion

PROS:

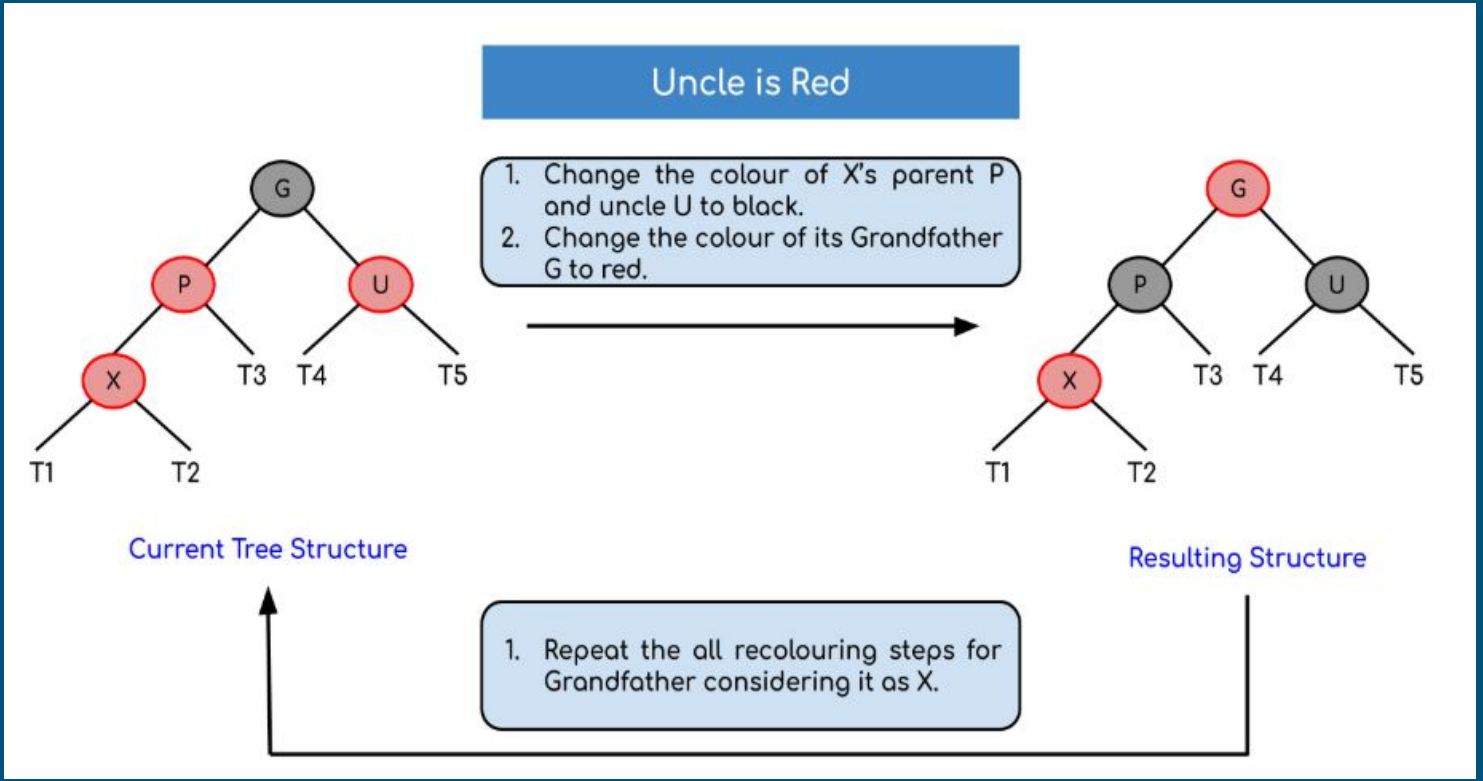
Insert is quite fast

Search is also fast

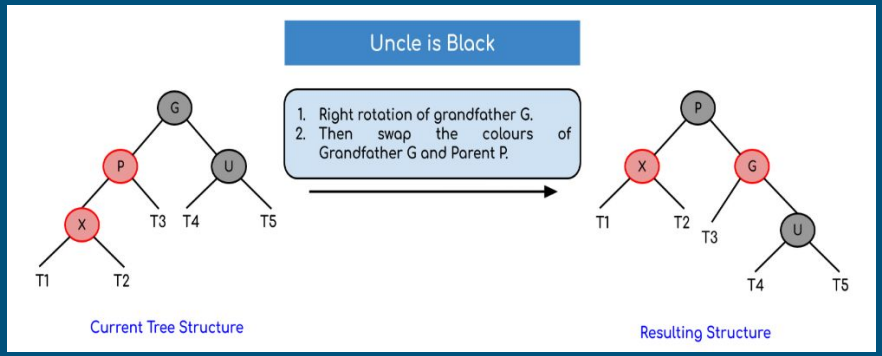
CONS:
Space complexity- 1 bit of data to store color (most data sets it won't matter but big sets of data it will)

Not as strictly balanced as AVL- search time is slower than AVL

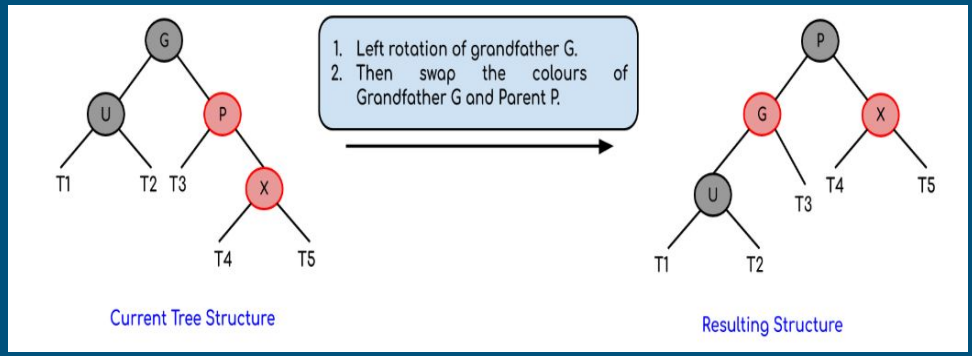
Recolor


Uncle is Red
1. Change the colour of X's parent P and uncle U to black.
2. Change the colour of its Grandfather G to red.
G
P
U
X
T1
T2
T3
T4
T5
Current Tree Structure
Resulting Structure
1. Repeat the all recolouring steps for Grandfather considering it as X.

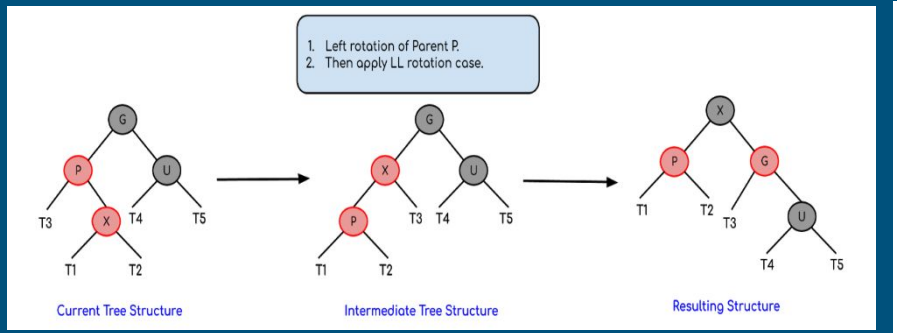
Rotate



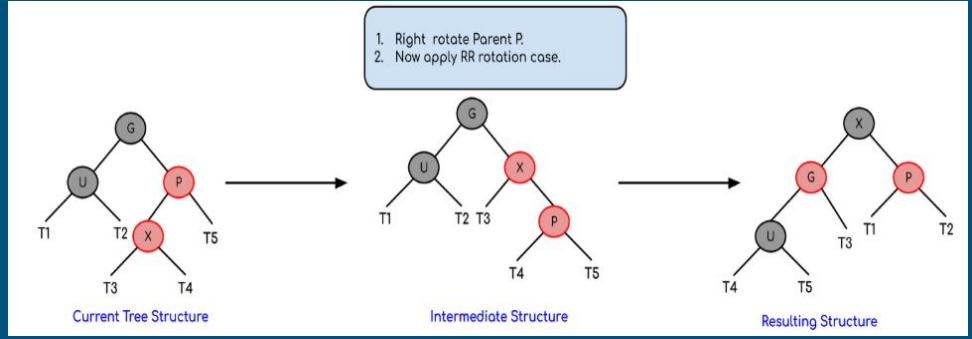
Left Left



Right Right

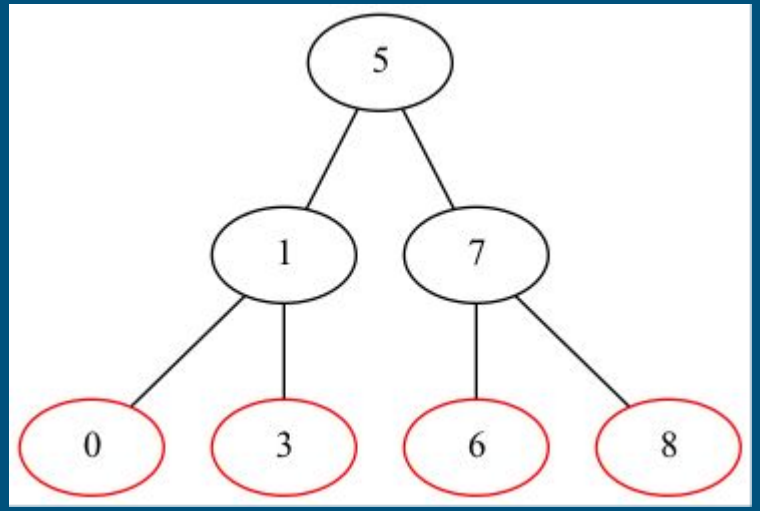


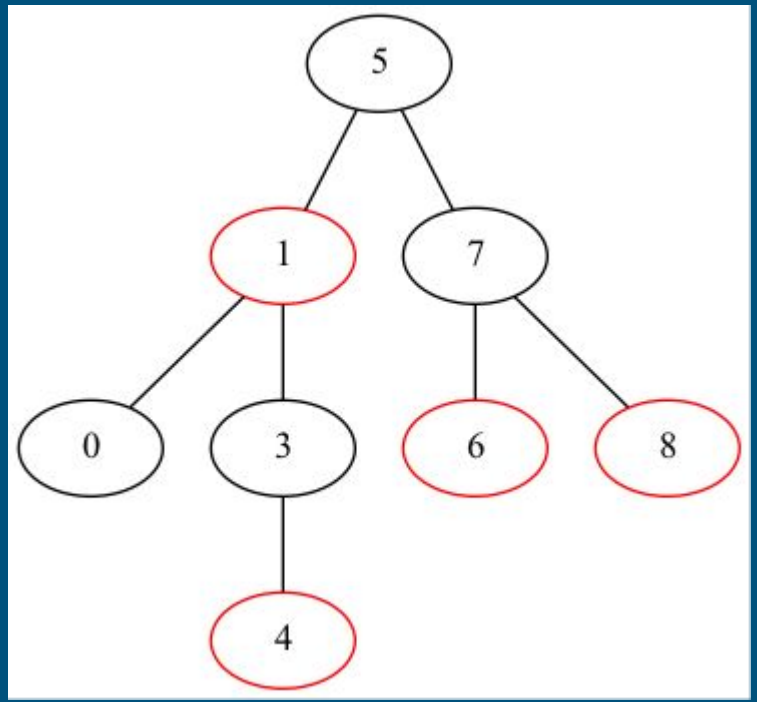
Left Right



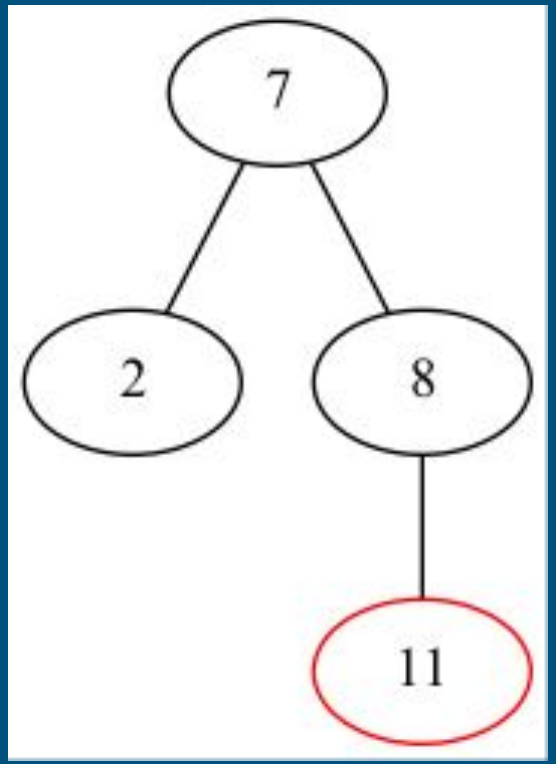
Right Left

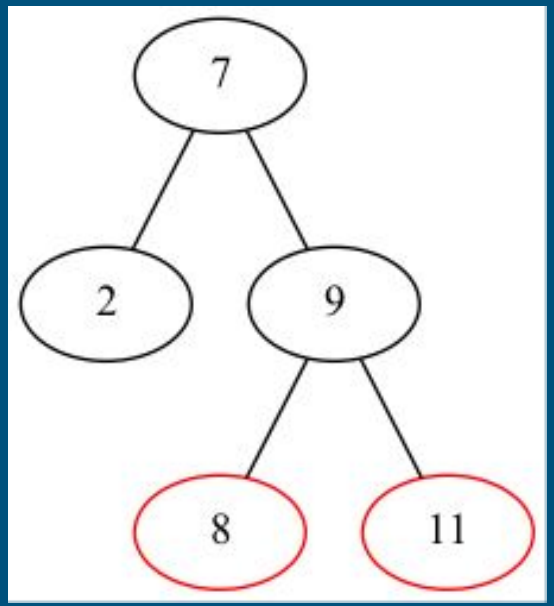
Red-Black Visualization Recolor





Red Black Visualization Rotate-RL





Implementation

```
class RBTree(BSTree):
    def __init__(self):
        super(RBTree,self).__init__()

    def add(self,key,value):
        #add it using super
        current=super(RBTree,self).add(key,value)
        current.rOb=1

        #if it is the root-change to black
        if current==self.root:
            current.rOb=0

        elif current.parent is not None and current.parent.rOb !=0:
            while(current is not self.root):
                #print(current.value)
                pibling=self.getpibling(current)


                if pibling is not None and pibling.rOb==1:
                        #print("red")
                        current=self.recolor(current,pibling)
                        #if there is no grandparent of current- if recolor doesn't return grandparent
                        #we recolored up to root and our work is done
                        if current is None:
                            break
                        #if current is root loop will end-check at end of loop
                else:#pibling is black
                    #print("black")
                        #current needs to be rotated around grandfather so
                    current=self.rotate(current)
                        ##if rotate happens-no need to loop up rest of tree so
                    break
                #if pibling IS none-dont continue looping
```

ScapeGoat Tree

- A type of self-balancing BSTree which does **not** use rotations

1. Uses BS to find the correct spot for a node
2. After insertion, recursively goes up every node in the correct side and checks to see if each node is balanced
3. If a node is unbalanced, we call this the “Scapegoat”
4. Remove the sub-tree starting at the scapegoat,
5. turn it into a sorted list
6. Rebuild into a BSTree and glue back on original tree

Implementation

```
scapegoat = self.find_scapegoat(node_to_insert)
if scapegoat is None:
    return

new_sub_root = self.rebalance(scapegoat)

#Assign the correct pointers to and from scapegoat
scapegoat.left_child = new_sub_root.left_child
scapegoat.right_child = new_sub_root.right_child
scapegoat.key = new_sub_root.key
scapegoat.value = new_sub_root.value
scapegoat.left_child.parent = scapegoat
scapegoat.right_child.parent = scapegoat
```

More Implementation

```
def find_scapegoat(self, node):
    if node == self.root:
        return None
    while self.is_balanced(node):
        if node == self.root:
            return None
        node = node.parent
    return node

def is_balanced(self, node):
    node_size = self.subtree_size(node.left_child) - self.subtree_size(node.right_child)
    if node_size <= 1 and node_size >= -1:
        return True
    return False

def subtree_size(self, node):
    #recursively calculates the size of the subtree of an input node
    if node == None:
        return 0
    return self.subtree_size(node.left_child) + self.subtree_size(node.right_child) + 1
```

Scapegoat Tree: More Discussion

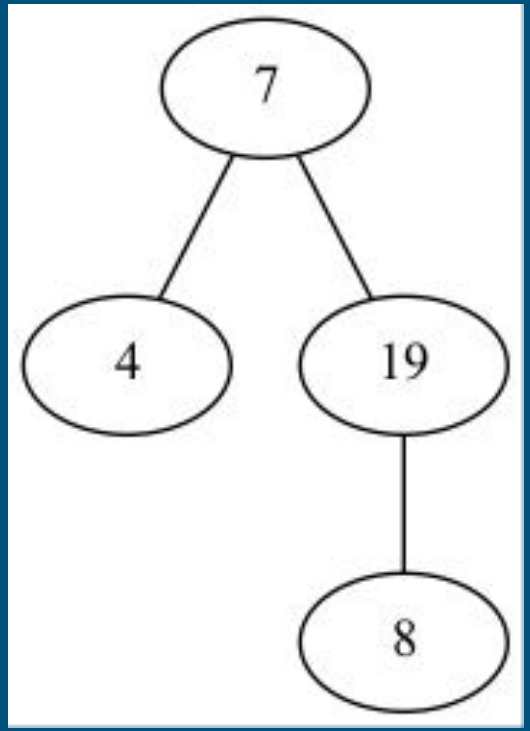
PROS

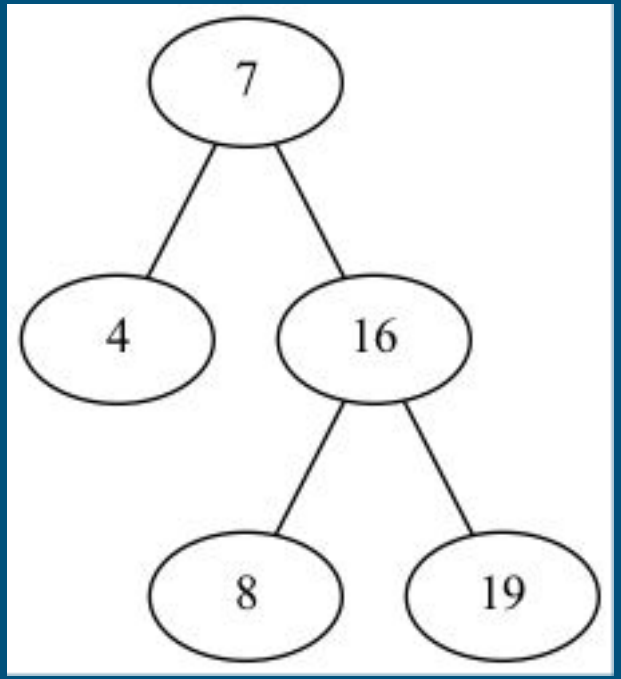
- Not too complicated to implement and understand
- Very space efficient! Only needs to store node's content (not height like AVL tree or color like red black)
- O(n log n) run time since the comparing operations are just sorting a list!

CONS

- Efficiency problems:
- If a tree is still balanced after insertion, then we still have to check every single node to verify this
- If a node is shallow in the tree, then sorting that list takes a long time

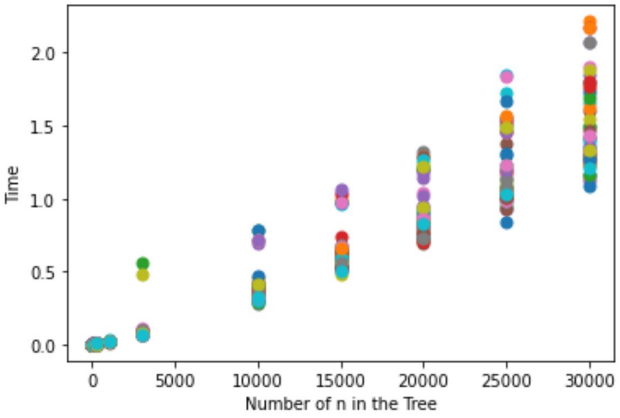
ScapeGoat Visualization





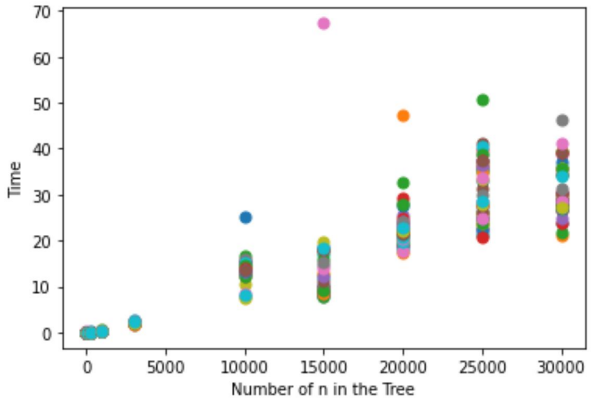
Results for Insertion

Out[24]: Text(0.5, 0, 'Number of n in the Tree')



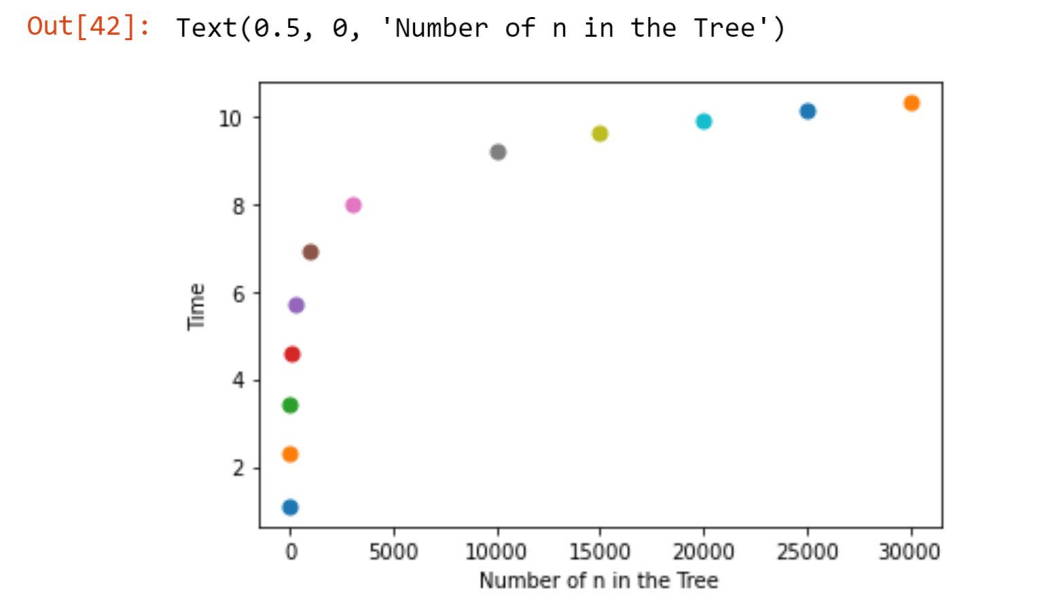
RedBlack

Out[29]: Text(0.5, 0, 'Number of n in the Tree')



ScapeGoat

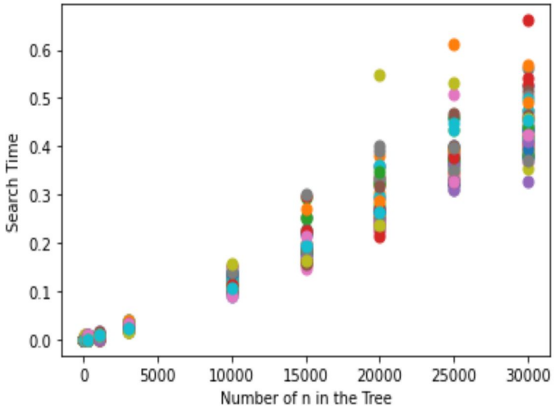
Theoretical Results Insertion



AVL Insertion O(logn)

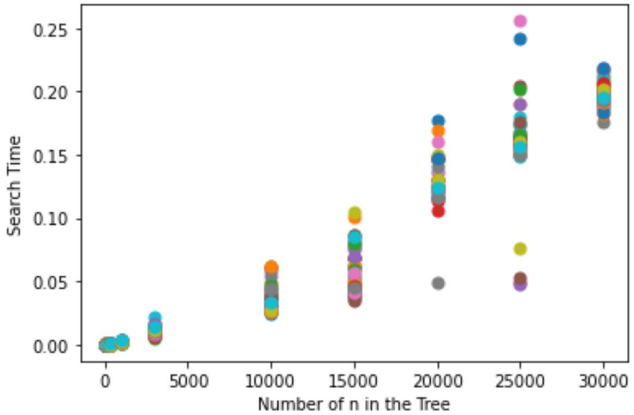
Results for Searching

Out[17]: Text(0.5, 0, 'Number of n in the Tree')



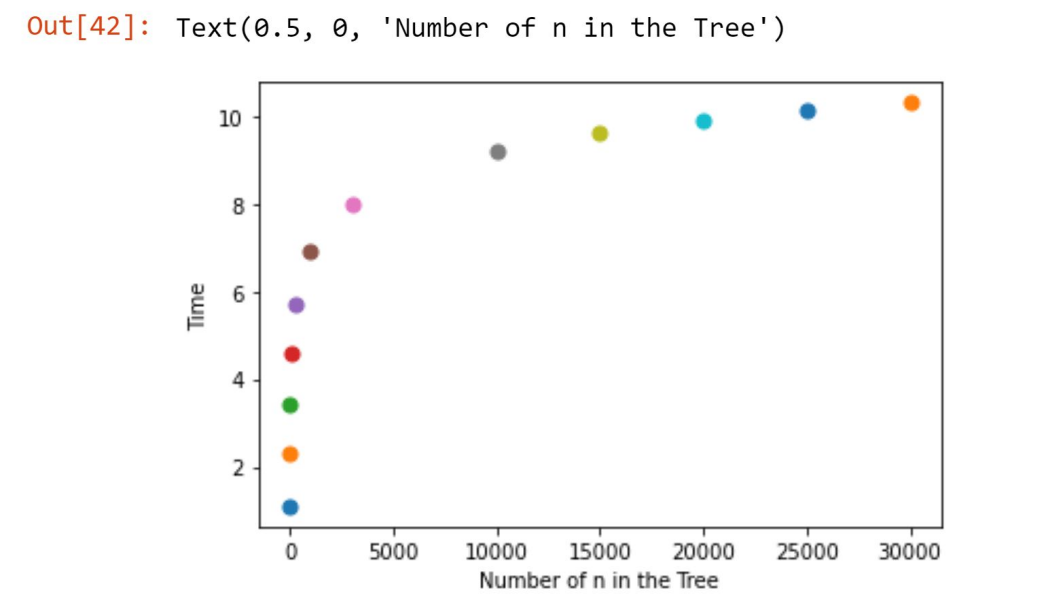
RedBlack

Out[25]: Text(0.5, 0, 'Number of n in the Tree')



ScapeGoat

Theoretical Results Searching



AVL Insertion O(logn)

Visualization

-Graphviz

vim g.gv
dot -Tpng g.gv -o scape2.png

```
my_tree_to_graphviz(tree)
```

tree

```
graph {
7
4
7 -- 4
16
7 -- 16
8
16 -- 8
19
16 -- 19
}
```

```
def my_tree_to_graphviz(tree):
    queue=[tree.root]
    print("graph {")
    while len(queue)>0:
        new_node =queue.pop(0)

        print(new_node.key)

        if new_node.parent is not None:
            #graphRb.edge(str(new_node.parent.key),str(new_node.key))
            print(new_node.parent.key,"--",new_node.key)
        if new_node.left_child is not None:
            queue.append(new_node.left_child)
        if new_node.right_child is not None:
            queue.append(new_node.right_child)
    print("}")
```

```
graph {
1 [color=black]
0 [color=black]
1 -- 0
4 [color=black]
1 -- 4
2 [color=red]
4 -- 2
16 [color=red]
4 -- 16
}
```

```python
def my_rbtree_to_graphviz(tree):
    queue=[tree.root]
    print("graph {")
    while len(queue)>0:
        new_node =queue.pop(0)
        if(new_node.rOb==0):
            #graphRb.attr('node', color='black')
            #graphRb.node(str(new_node.key))
            print(new_node.key,"[color=black]")
        else:
            #graphRb.attr('node', color='red')
            #graphRb.node(str(new_node.key))
            print(new_node.key,"[color=red]")

        if new_node.parent is not None:
            #graphRb.edge(str(new_node.parent.key),str(new_node.key))
            print(new_node.parent.key,"--",new_node.key)
        if new_node.left_child is not None:
            queue.append(new_node.left_child)
        if new_node.right_child is not None:
            queue.append(new_node.right_child)
    print("}")
```

To Answer Our Questions

- Which is faster for insert: ScapeGoat
- Which is faster for search: RedBlack
- Which has fewer rebalances: RedBlack
- Which was harder to implement: ScapeGoat
- AVL Tree
  - Insert: O(logn)
  - Search: O(logn) worst case O(n)
  - Rotations: O(n)
- RB Tree:
  - Insert: O(logn)
  - Search: O(logn) worst case O(n)
  - Rotations: O(n)
- Scapegoat:
  - Insert: O(nlogn)
  - Search: O(n)
  - Rotations: O(n)