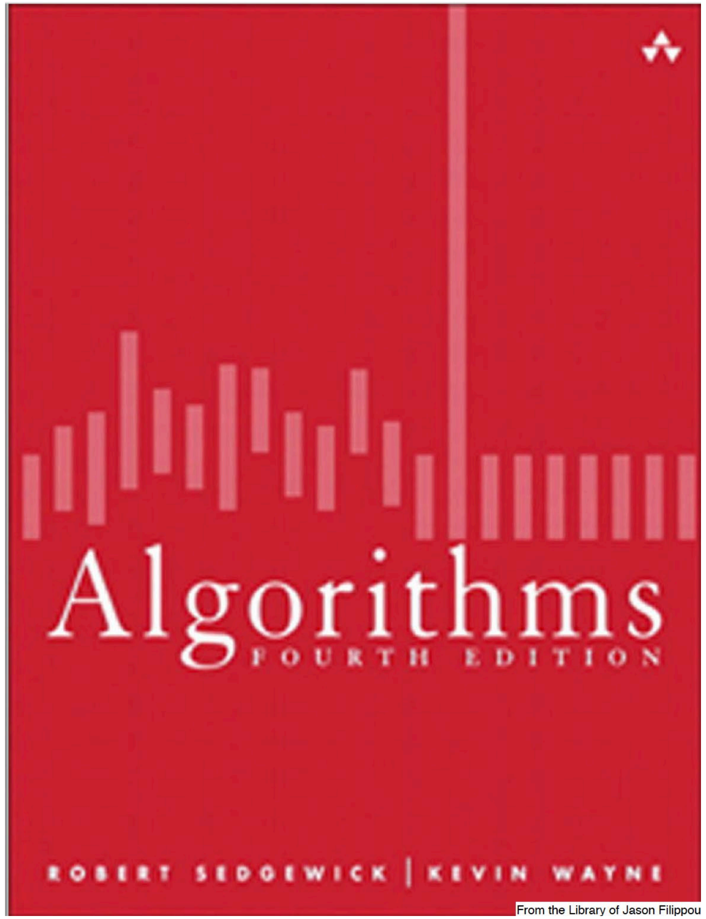
Red-Black Trees

CMSC 420

Resources

- We will be discussing “left-leaning” Red-Black Binary Search Trees.
- Sedgewick & Wayne are a great resource (chapter 3.3).
- Slides’ programming model based on theirs.



java.util.TreeMap

compact1, compact2, compact3
java.util

Class TreeMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.TreeMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Implemented Interfaces:

`Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>`

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

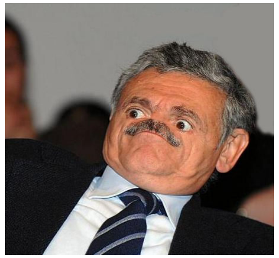
This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

The Treemap

compact1, compact2, compact3
java.util

Class TreeMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.TreeMap<K,V>



Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

Utility

- We will implement 2-3 trees with a binary tree!
- So we get **search** and the top-down parts of insertion and deletion for free.
  - So we also get the top-down parts of insertion and deletion for free!
- Idea: We will use red links to represent 3-nodes, and black links to represent the regular links between the nodes in a 2-3-Tree.
  - It helps Jason to imagine those as some sort of superglue.

Caveats

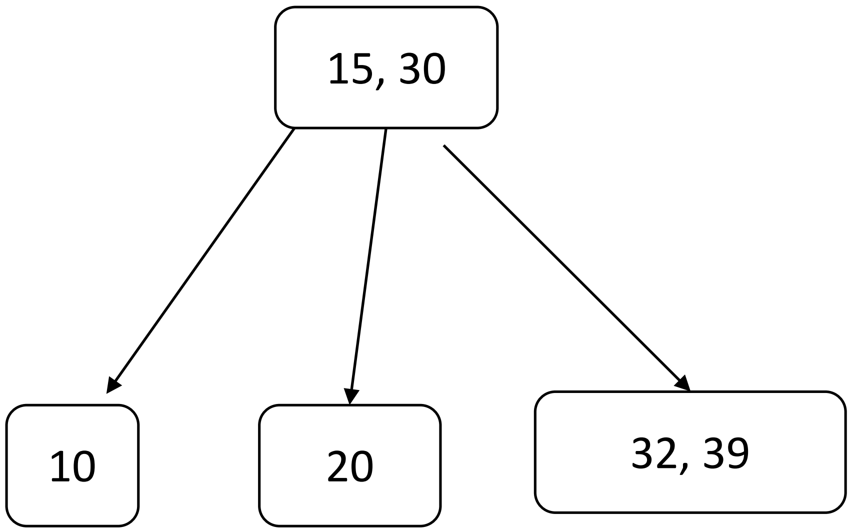
1. We will be talking about so-called “Left-leaning” Red-Black Trees.
   - Those are easier to implement than “traditional” Red-Black Trees and maintain the same theoretical properties.
2. Red-Black Trees do **not** implement 2-3 tree key rotations!
   - That is, when a node overflows or underflows, the relevant 2-3 tree would “explode” (resp. “vanish”) the node without looking at its siblings for space to hold a key.

Caveats

3. We will **not** talk about "hard" deletion from a Red-Black Tree.
    - "Soft" deletion: Also known as "Mark-and-sweep". Mark nodes "dead" by setting a bit, and at periodic intervals, build a new tree by traversing the existing tree and inserting only the "live" nodes into the new tree.
        - ArrayLists do mark-and-sweep!
    - "Hard" deletion: our familiar semantics: actual key or node removal from the structure.
        - Very hard in RBBSTs, and actually allows for key rotations (which we have learned in 2-3 trees)

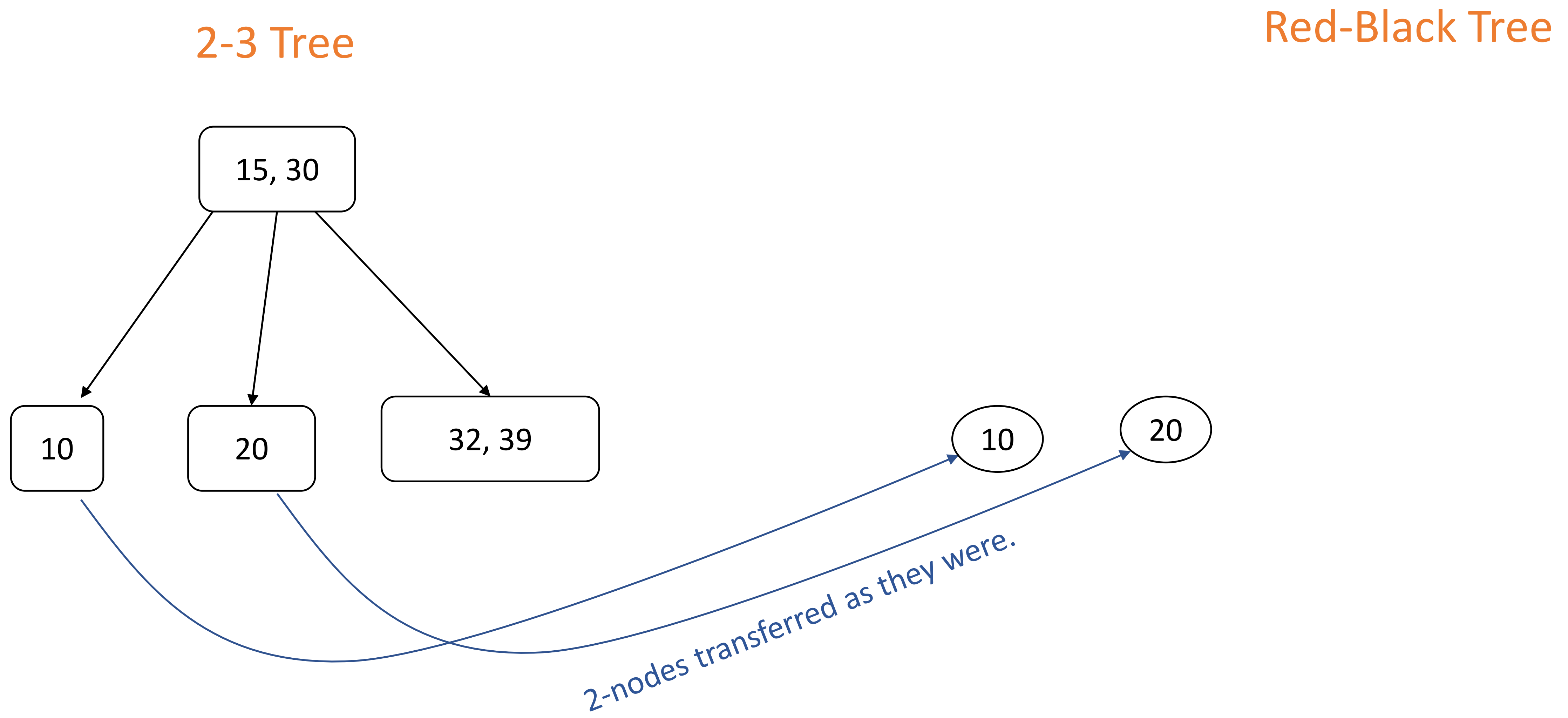
Representation

2-3 Tree

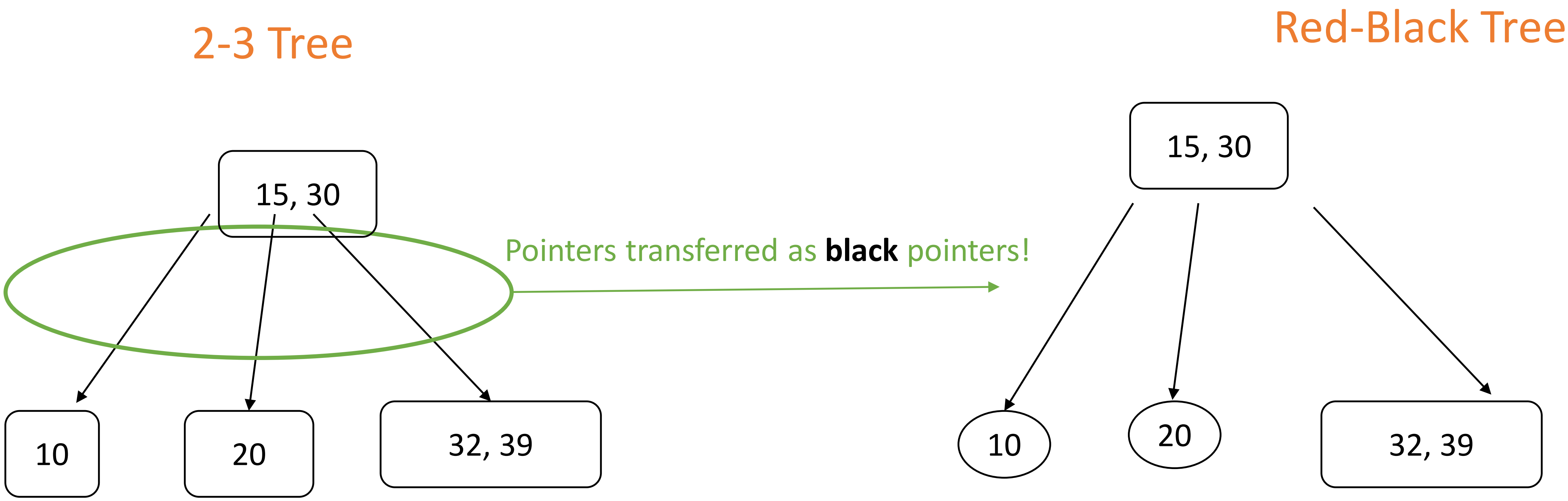


Red-Black Tree

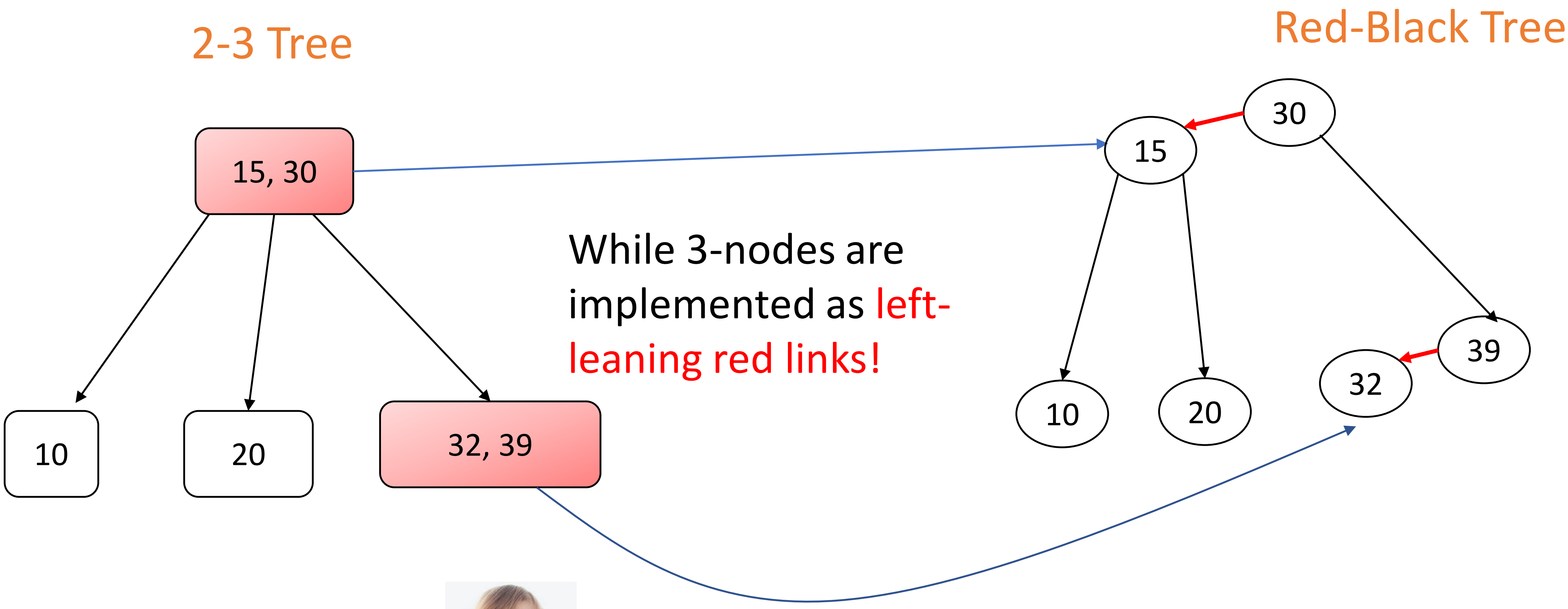
Representation



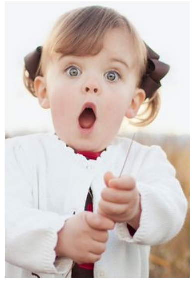
Representation



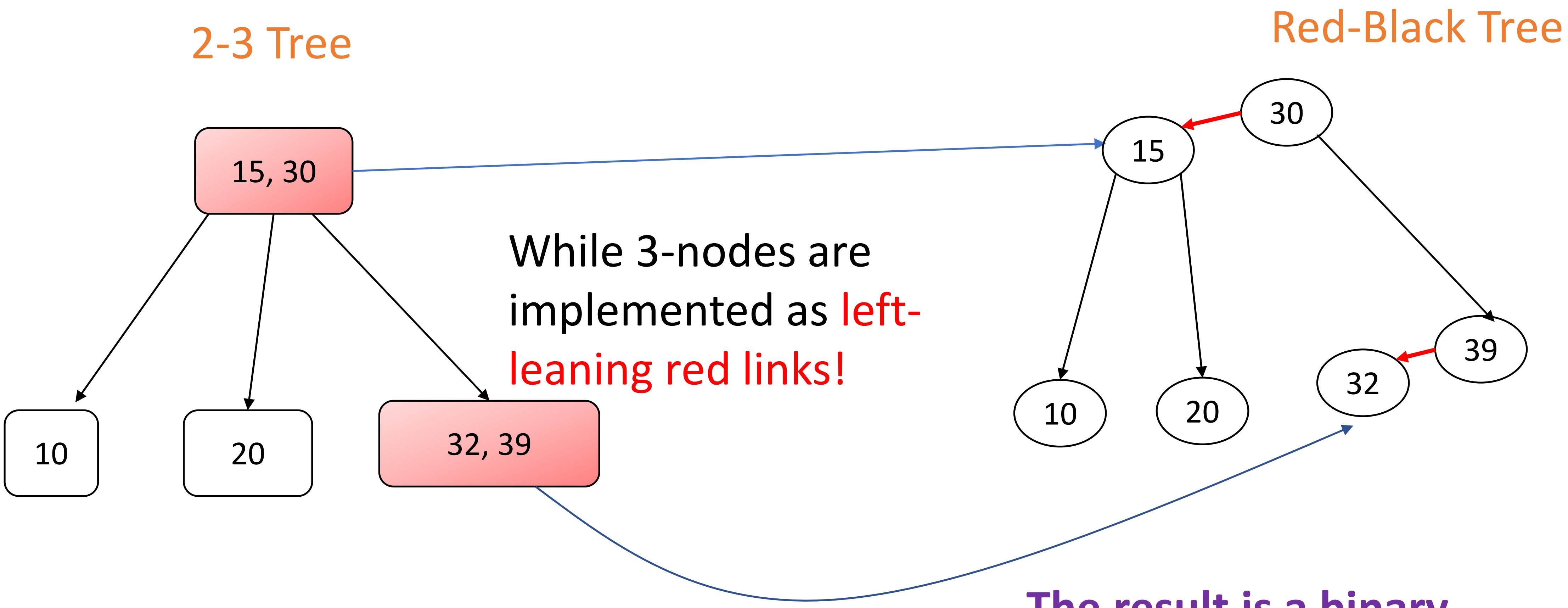
Representation



While 3-nodes are implemented as left-leaning red links!



Representation

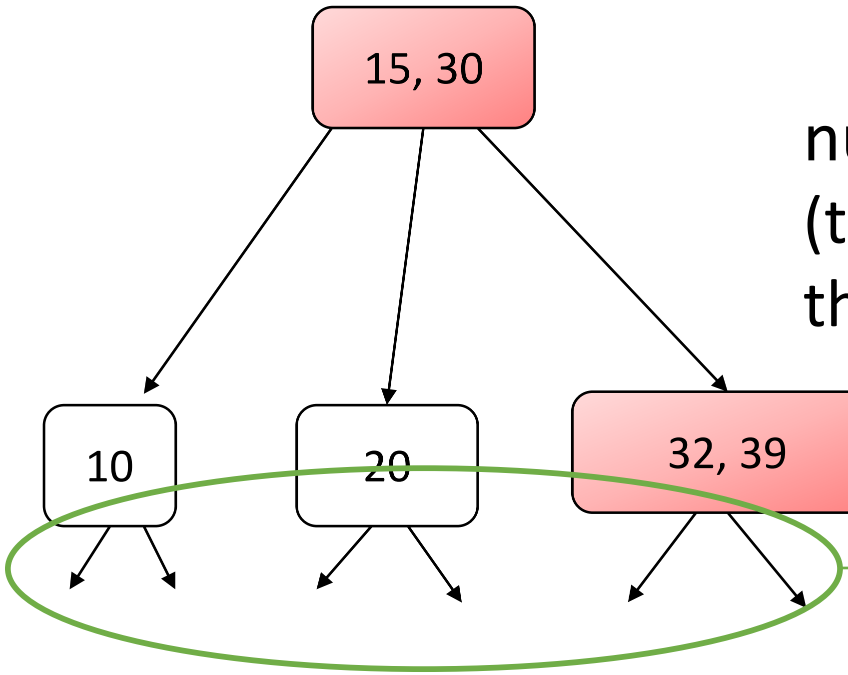


While 3-nodes are implemented as left-leaning red links!

The result is a binary tree over keys!

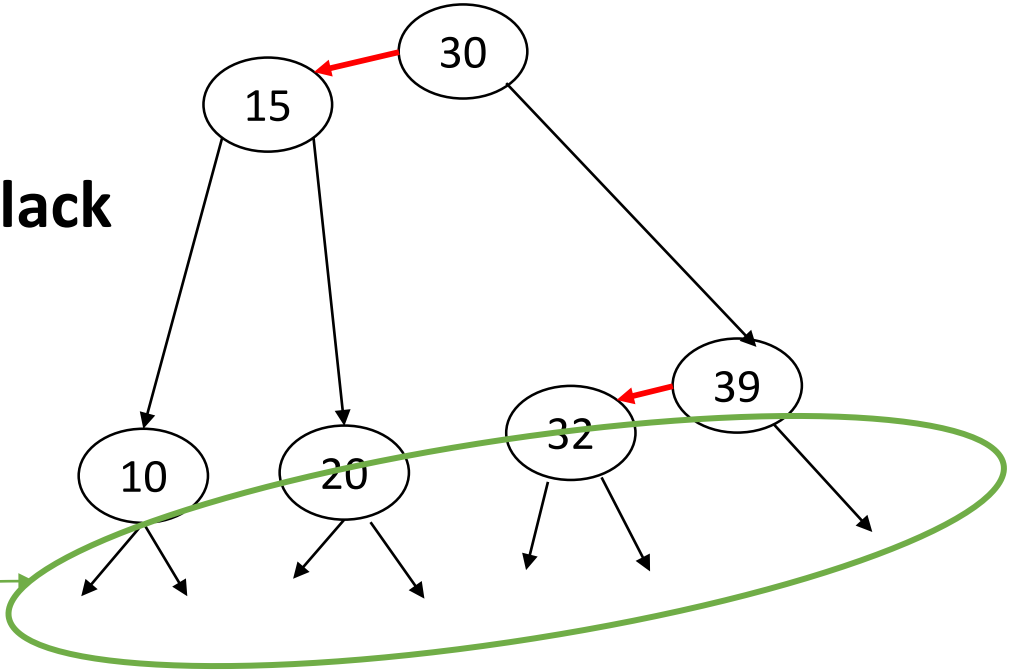
Representation

2-3 Tree

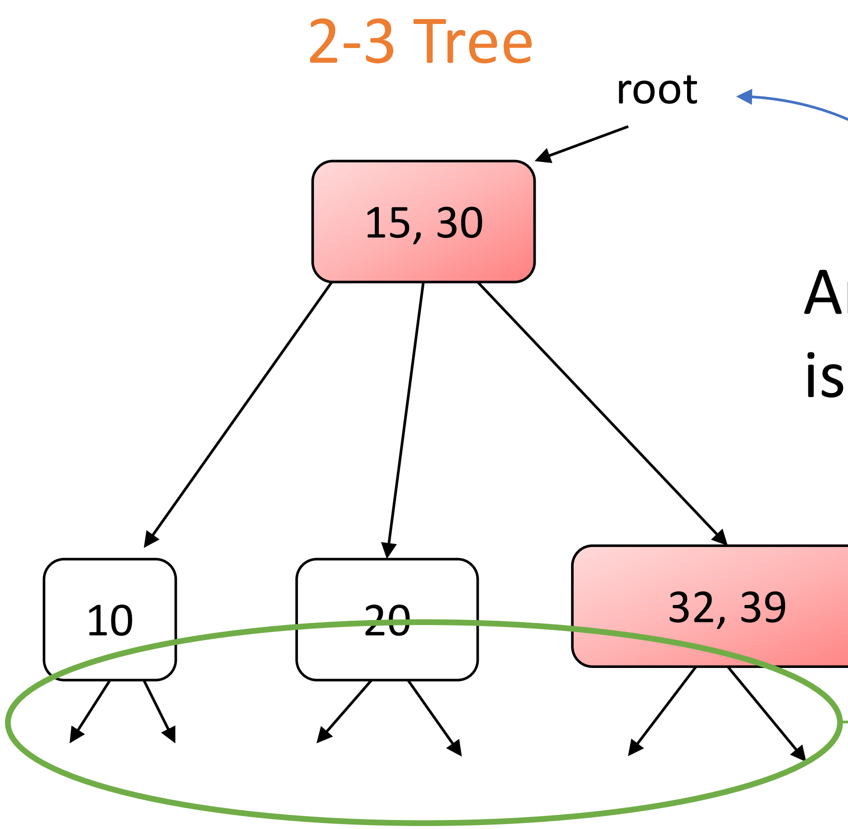


`null` links are **also black** (they exist, unlike in threaded BSTs)

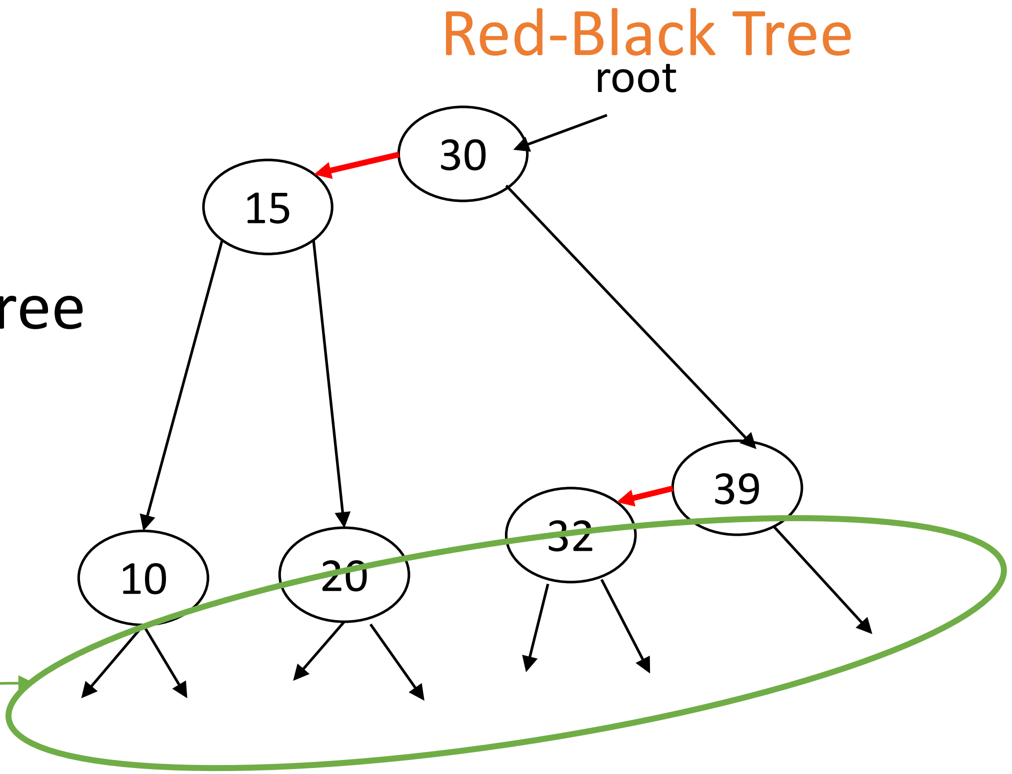
Red-Black Tree



Representation



And the root of the tree is also **black.**



An immediate benefit of RBBSTs

An immediate benefit of RBBSTs

- You can use existing BST code with virtually no modification!
  - **Particularly for searches!**
- In fact, it makes a lot of sense to override an existing BST class
  - You would then need to make any inner node classes in the BST class `protected`

An immediate drawback

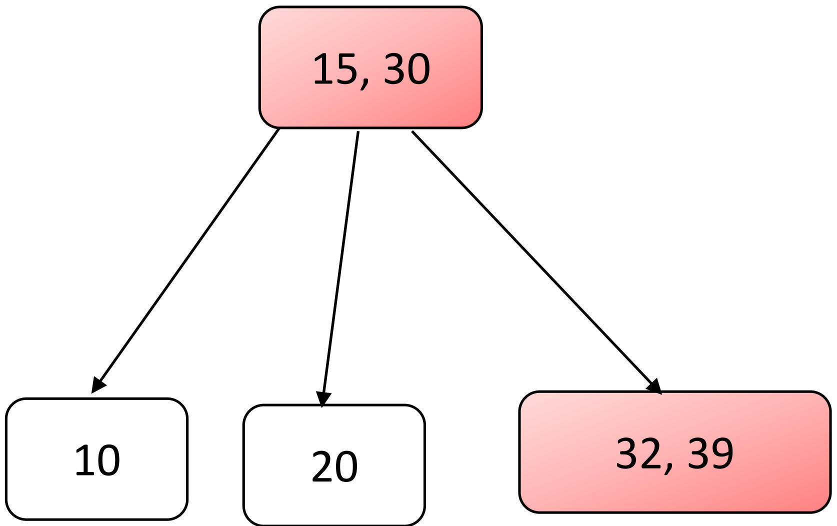
- Red links make our tree taller ☹.
- Compared to AVL Trees, which have height in $[\log_2 n\,,\ \ \log_2 n + 1]$
- Red-Black Trees have a height in $[\log_2 n\,,\ \ 2 \cdot \log_2 n]$
- Best case: A 2-3 tree with only "two-nodes" (so, a BST) will lead to a perfectly balanced binary tree in the RBBST representation.
- Worst case: A 2-3 tree whose left links are all red.
  - That'll lead to $2 \cdot \log_2 \mathrm{n}$.

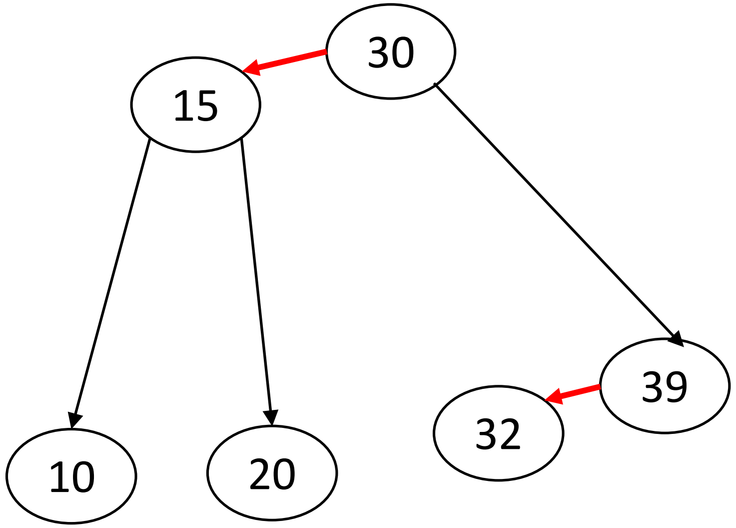
Quiz for you

- In a red-black tree, we are guaranteed at least one node n who has two red links connected to it.

True

False





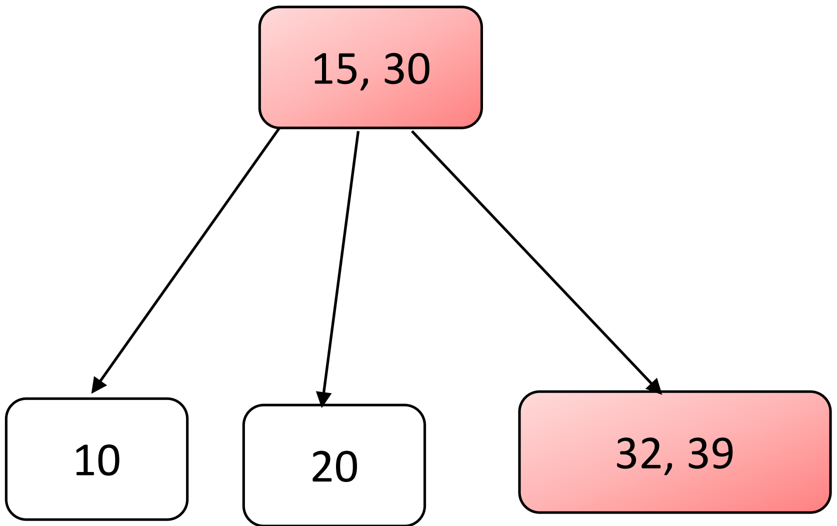
Quiz for you

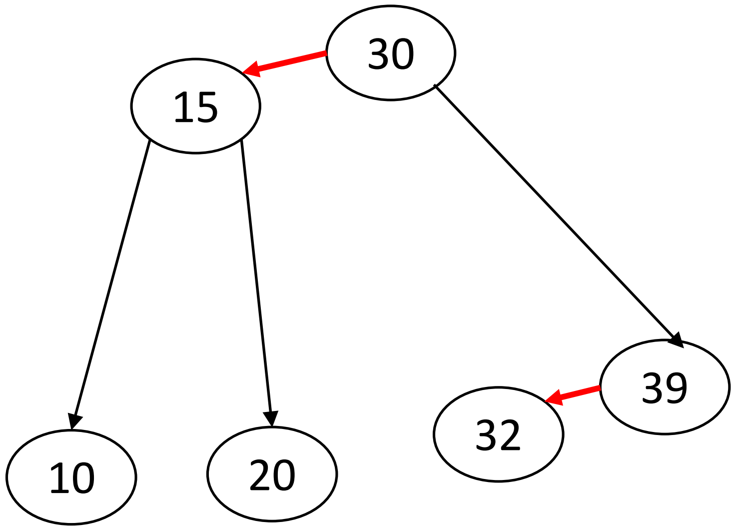
- In a red-black tree, we are guaranteed at least one node n who has two red links connected to it.

True

False

In fact, we have the **opposite guarantee**: there are ***no nodes*** that are linked to by 2 red links.



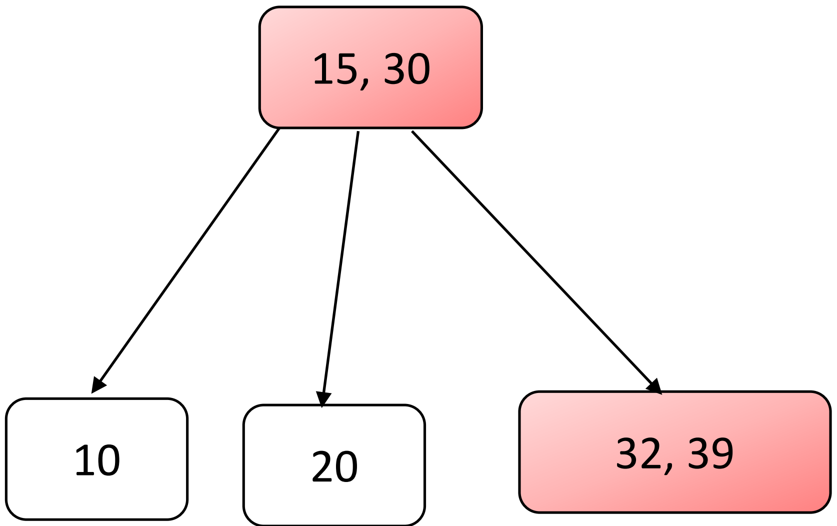


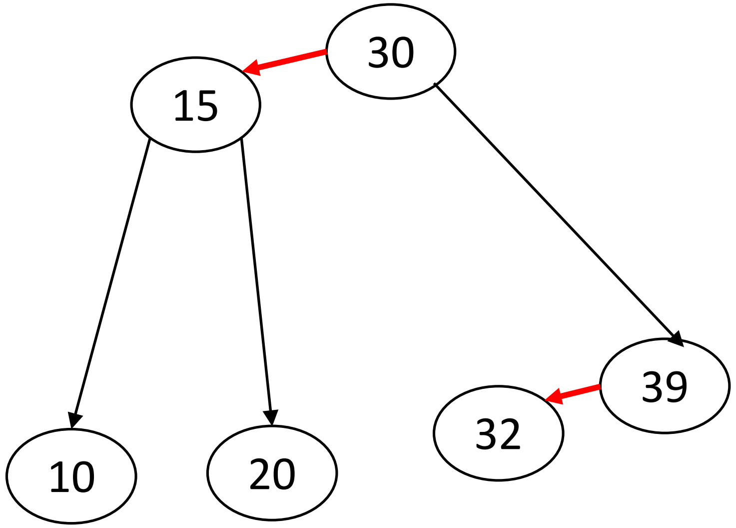
Another quiz for you

- In a (left-leaning) red-black tree, it is guaranteed that at least one red link ℓ is **right-leaning** instead of **left-leaning**

True

False





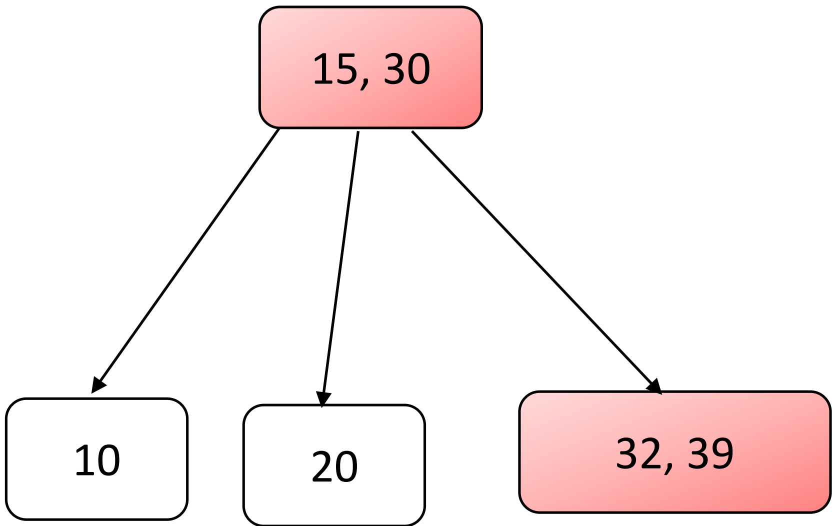
Another quiz for you

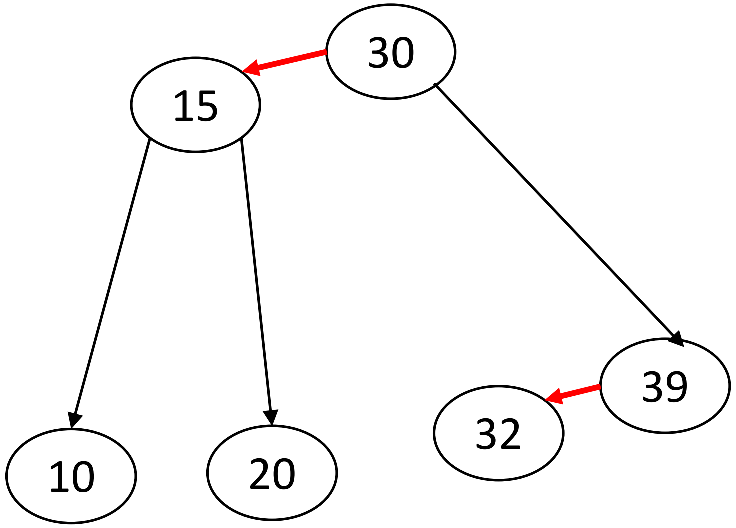
- In a (left-leaning) red-black tree, it is guaranteed that at least one red link ℓ is **right-leaning** instead of **left-leaning**

True

False

No right-leaning red links in an RBBST!





Node structure

- We need:
  - Links to our children
  - Data field (some `Comparable` type `T`)
  - A flag that tells us what **the color of the link from our parent to the current node is!**
  - For its implementation to be visible in subclasses

Node structure

- We need:
  - Links to our children
  - Data field (some Comparable type T)
  - A flag that tells us what **the color of the link from our parent to the current node is!**
  - For its implementation to be visible in subclasses

```
protected enum  Color{
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
     Color color;
}
```

Node structure

- We need:
  - Links to our children
  - Data field (some `Comparable` type T)
  - A flag that tells us what **the color of the link from our parent to the current node is!**
  - For its implementation to be visible in subclasses

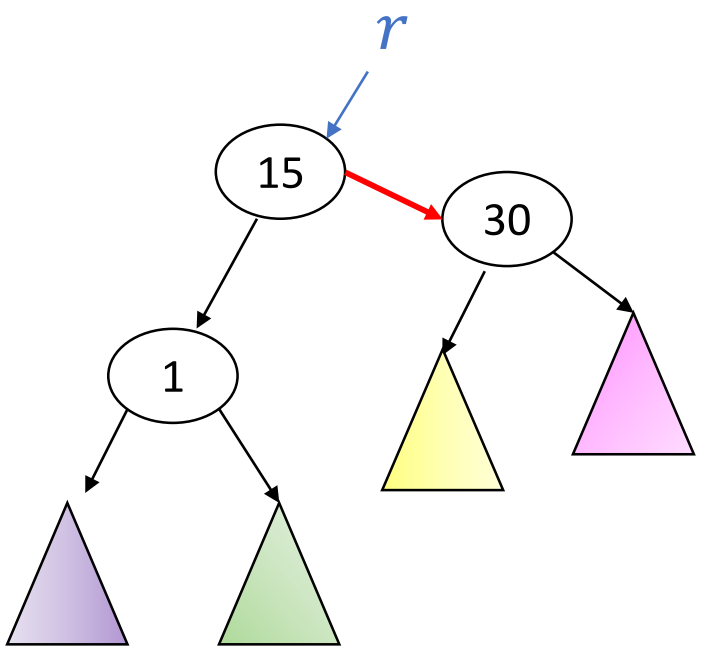
```
protected enum  Color{
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
     Color color;
}
```

This could work! ☺

Implement a utility

- During our operations, we will sometimes have right-leaning red links temporarily.

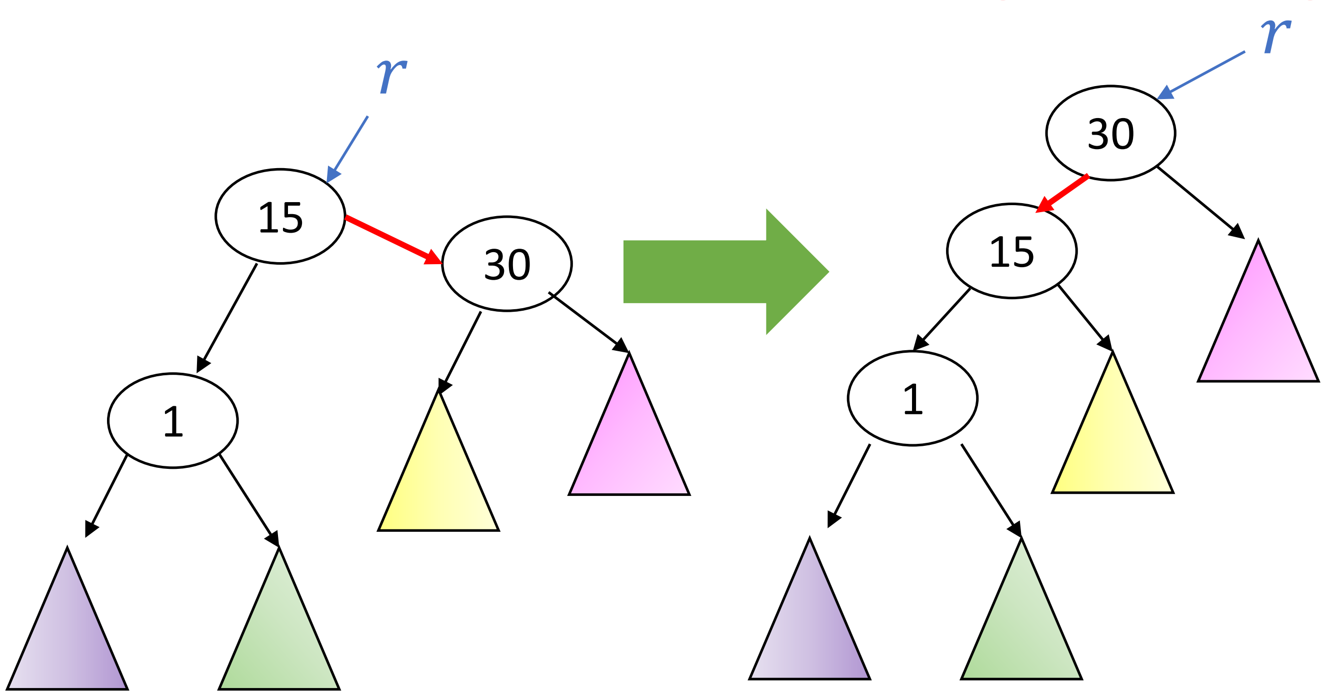
```
protected enum Color {
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
     Color color;
}
```



Implement a utility

- During our operations, we will sometimes have right-leaning red links temporarily.

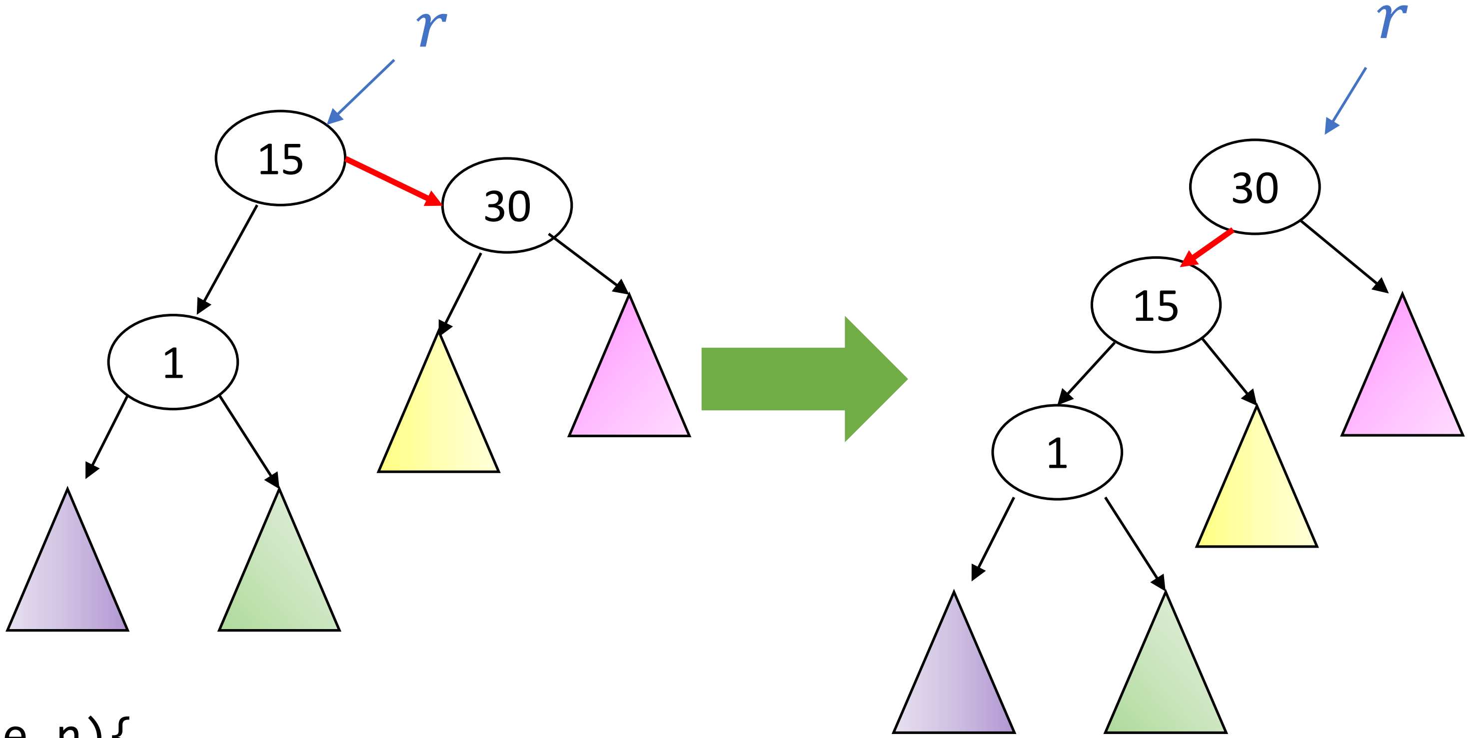
```
protected enum Color {
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
     Color color;
}
```



In your notes, write a private method called `rotateLeft(Node n)` such that when we call `r = rotateLeft(r)`, the result is as shown on the right!

Implement a utility

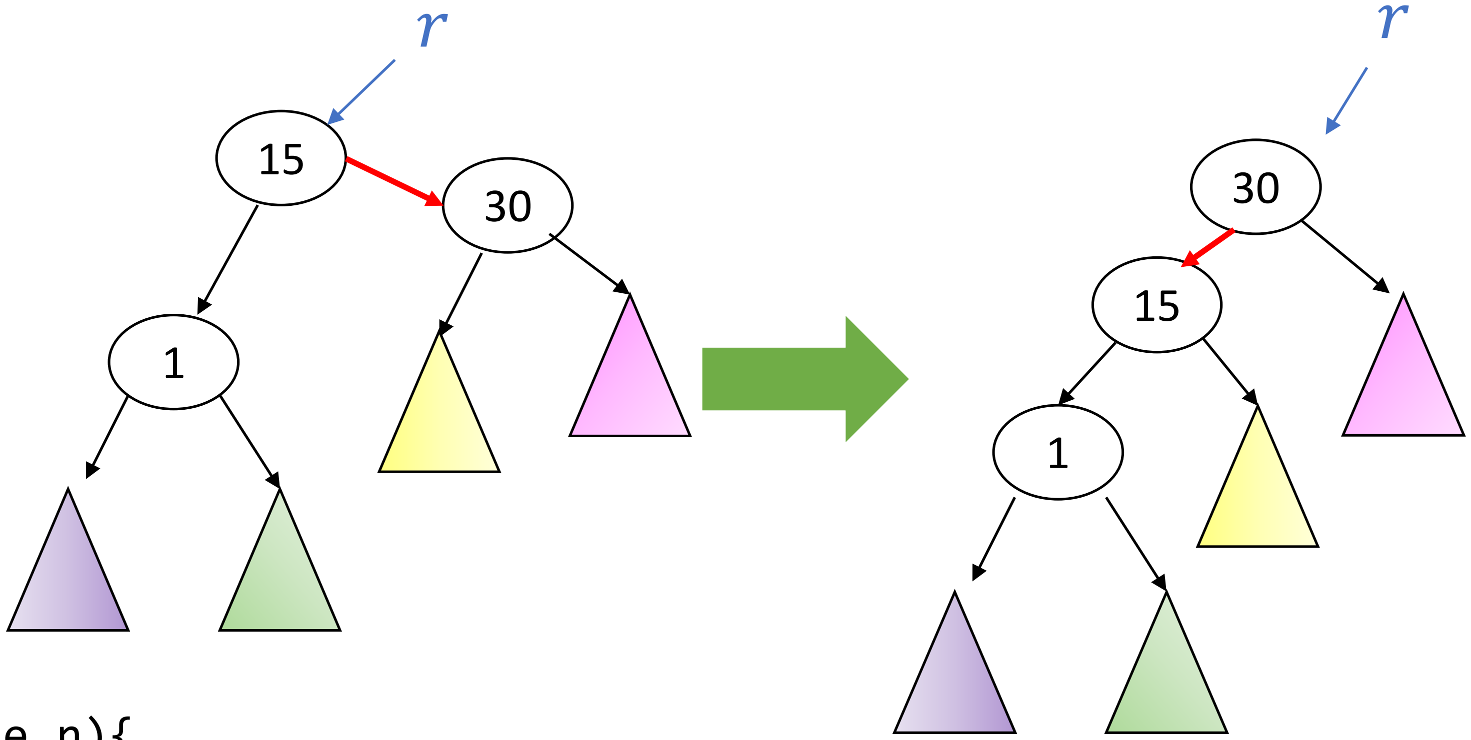
```
private enum  Color{
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
    Color color;
}
```




```
private Node rotateLeft(Node n){
    Node temp = n.right;
    n.right = temp.left; // How do I know that temp != null?
    temp.left = n;
    temp.left.color = RED;
    return temp;
}
```

Implement a utility

```
private enum  Color{
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
    Color color;
}
```



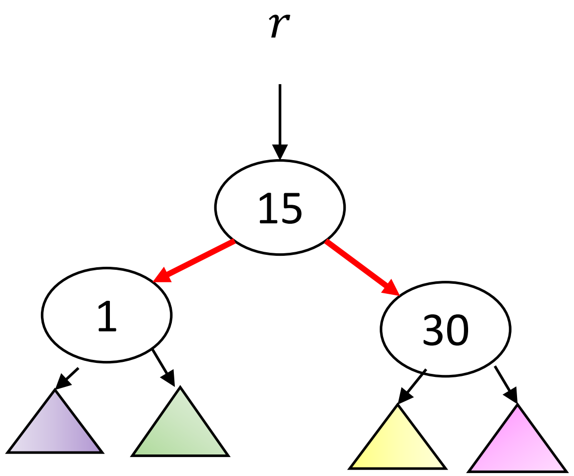

```
private Node rotateLeft(Node n){
    Node temp = n.right;
    n.right = temp.left; // How do I know that temp != null?
    temp.left = n;
    temp.left.color = RED;
    return temp;
}
```

rotateRight() symmetric.

Implement another utility

- Sometimes, during our operations, we will end up with a node connected to two red links.

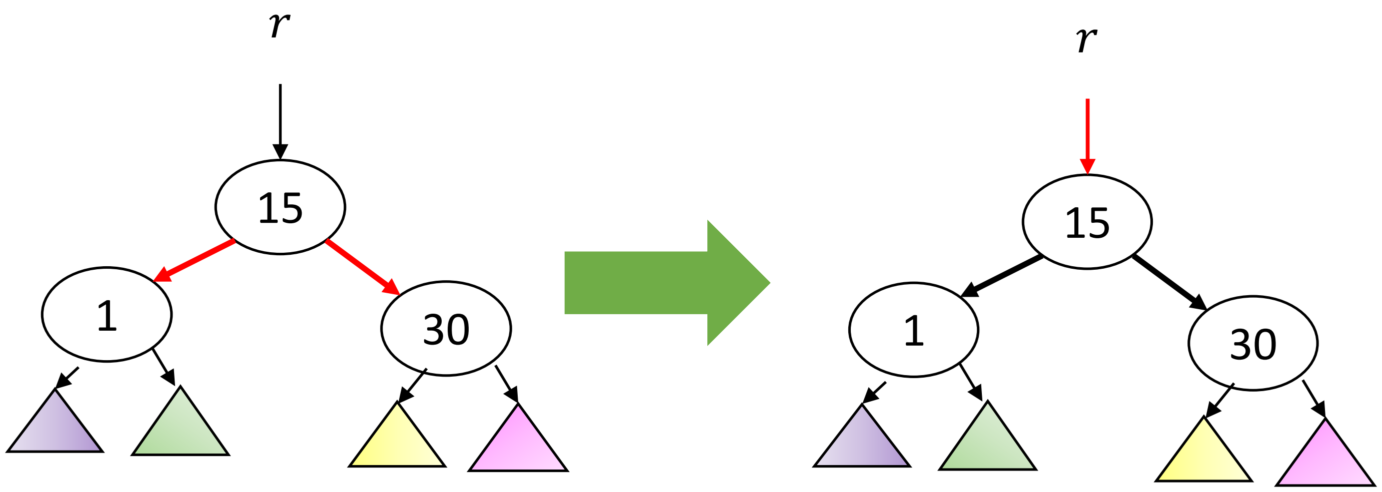
```
protected enum  Color{
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
    Color color;
}
```



Implement another utility

- Sometimes, during our operations, we will end up with a node connected to two red links.

```
protected enum  Color{
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
    Color color;
}
```

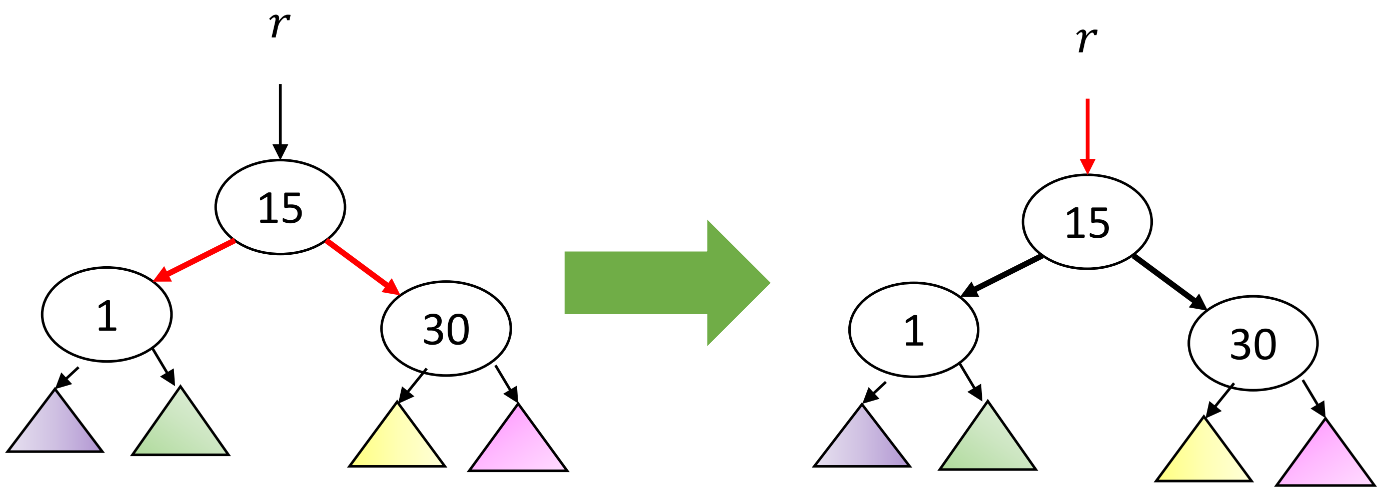


Write a void Java routine called `flipColors` which, when called as `flipColors(r)`, will transform the colors of the tree rooted at r as shown above!

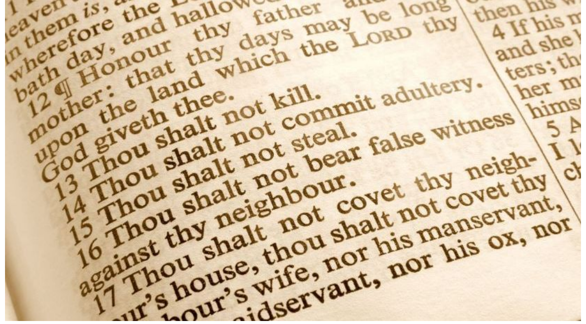
Implement another utility

- Sometimes, during our operations, we will end up with a node connected to two red links.

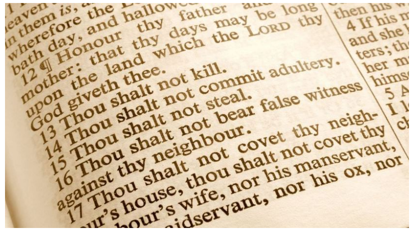
```
protected enum  Color{
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
    Color color;
}

protected void flipColors(Node n){
   n.left.color = n.right.color = BLACK;
   n.color = RED;
}
```



Red-Black Tree Invariants



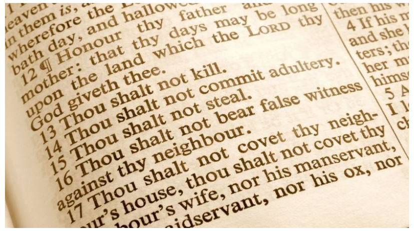
1. Root is always **Black**
2. All red links point to **left**
3. Any given node has at most one red link only (either **to** it or **from** it)
   - Only when we model 2-3 trees.
4. Perfect **black link** balance: For **any** given leaf node n, all paths from the root to one such node have the same number of **black** links.
   - This shouldn't be surprising, since **black** links correspond **exactly** to the edges of a 2-3 tree!



Maintaining our invariants

1. Root is always **Black**
   - After **every** insertion, we will make sure the last call will be explicitly making the root **black**!
2. All red links point to **left**
   - If we create a fresh node which is connected to its parent with a red link, but it's the parent's right child, we will rotate the parent left to make the red link point to the left

Maintaining our invariants



3. Any given node has one red link only (either **to** it or **from** it)
   - ➢ We will see **three ways** to rectify this. Recall the following:
     a) To have a node with more than one red link means that we have a 4-node, which **must** be dealt with!
     b) Only way to deal with this node is to split it, since Red-Black Trees cannot implement key rotations (the relevant 2-3 tree “splits” aggressively).

4. Perfect **black link** balance: For **any** given leaf node n, all paths from the root to one such node have the same number of **black** links.
   - If this is somehow violated, **we have a bug in our implementation** ☹

Insertion

2-3 tree

Red-**Black** tree

Insertion

2-3 tree

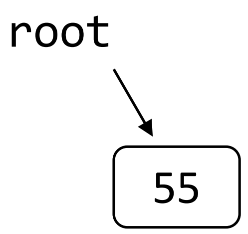
root $\rightarrow$ **X**

Red-Black tree

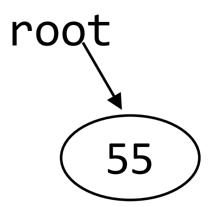
root $\rightarrow$ **X**

Insertion

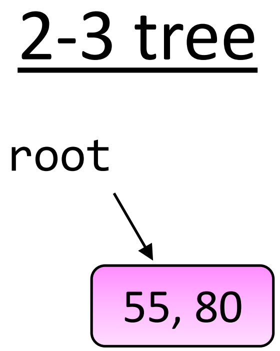
2-3 tree

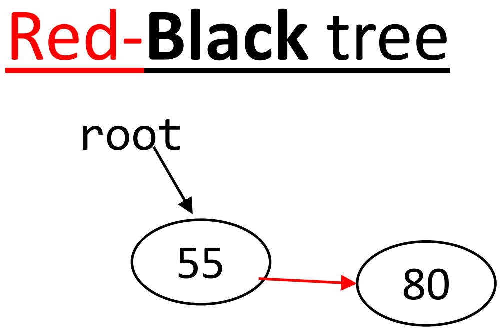


Red-**Black** tree



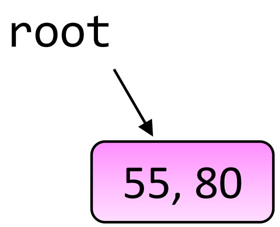
Insertion



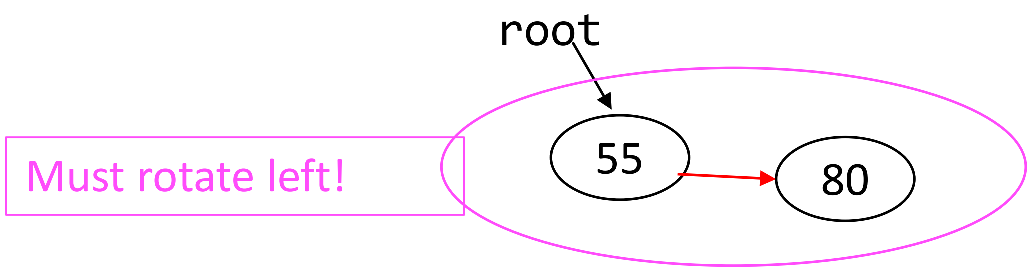


Insertion

2-3 tree

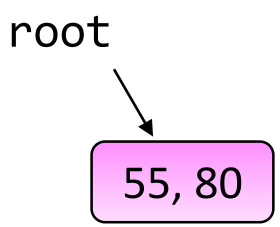


Red-**Black** tree

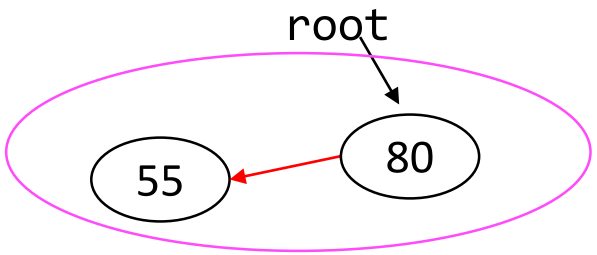


Insertion

2-3 tree

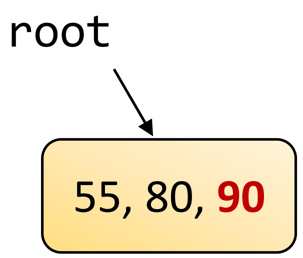


Red-Black tree

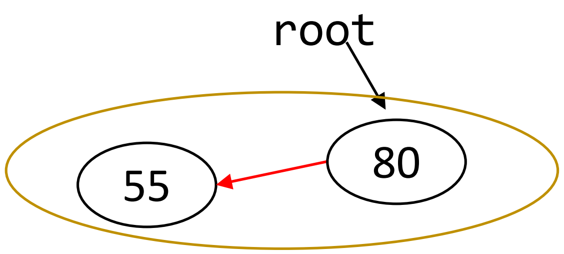


Insertion

2-3 tree

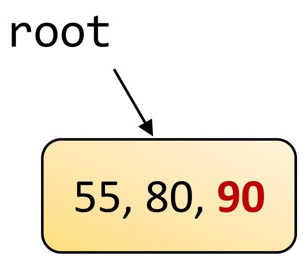


Red-Black tree

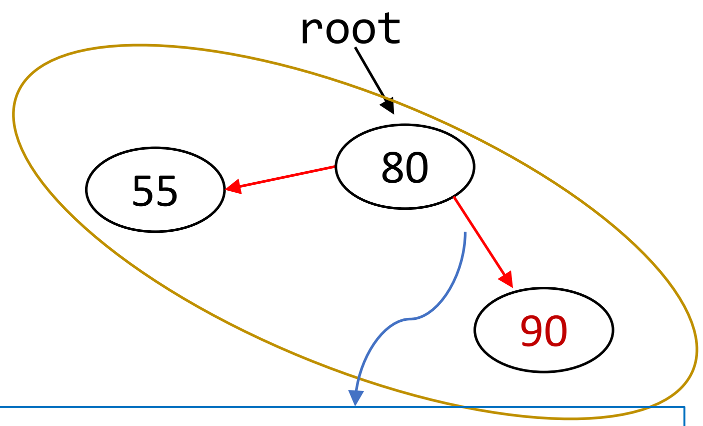


Insertion

2-3 tree



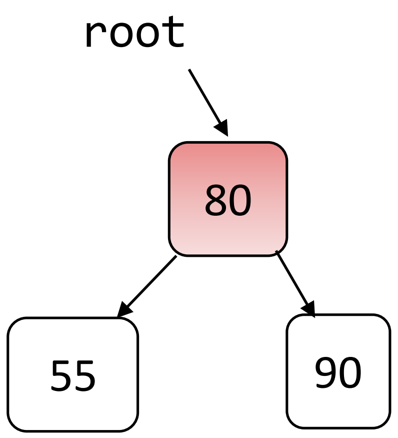
Red-Black tree



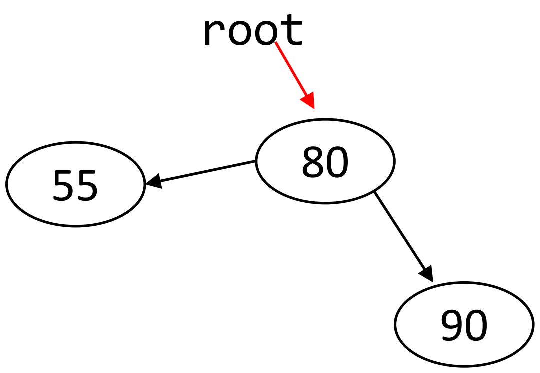
We always connect fresh nodes to their parents with a red link!

Insertion

2-3 tree



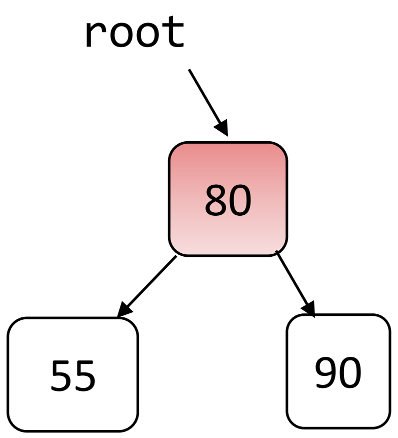
Red-**Black** tree



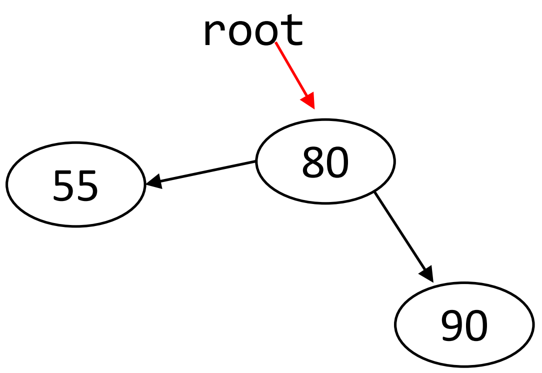
`flipcolors()` is used to exchange the link colors to represent the new 2-nodes…

Insertion

2-3 tree



Red-**Black** tree

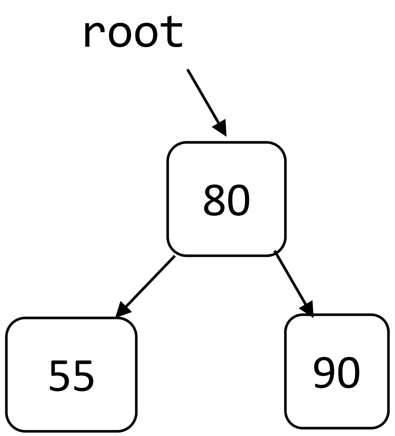


`flipcolors()` is used to exchange the link colors to represent the new 2-nodes…

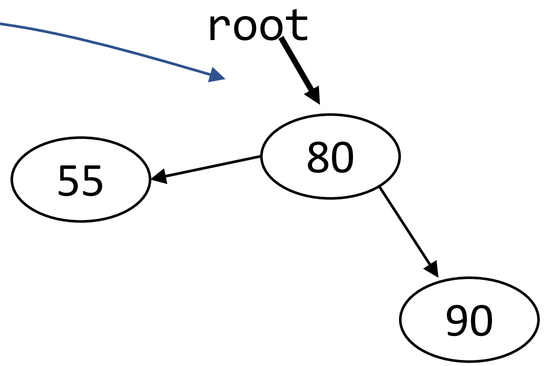
But the root node of an RBBST can never really be **red**, so we explicitly make it **black** at that level of the recursion.

Insertion

2-3 tree



Red-Black tree

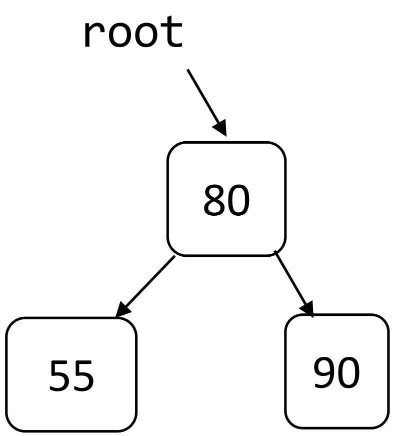


`flipcolors()` is used to exchange the link colors to represent the new 2-nodes…

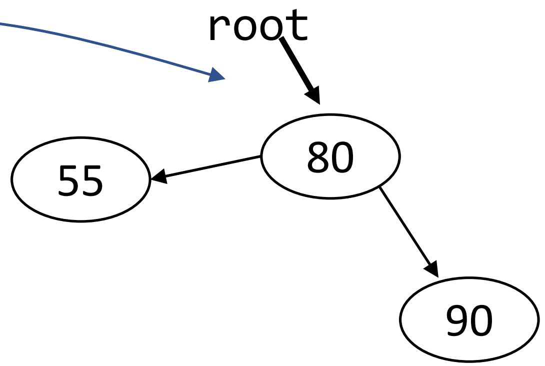
But the root node of an RBBST can never really be **red**, so we explicitly make it **black** at that level of the recursion.

Insertion

2-3 tree



Red-Black tree



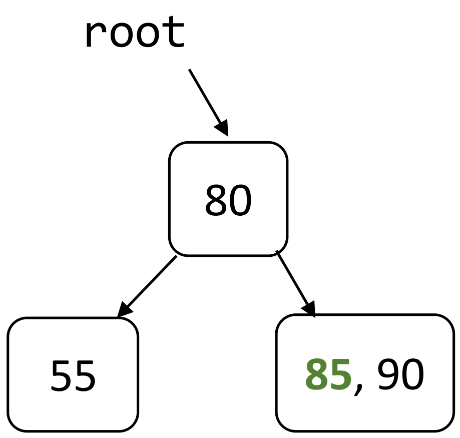

```
public void insert(Key key){
    root = insert(root, key);
    root.color = BLACK;
}
```

`flipcolors()` is used to exchange the link colors to represent the new 2-nodes…

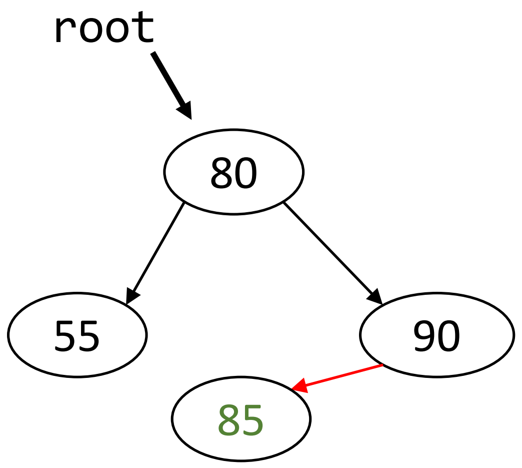
But the root node of an RBBST can never really be **red**, so we explicitly make it **black** at that level of the recursion.

Insertion

2-3 tree

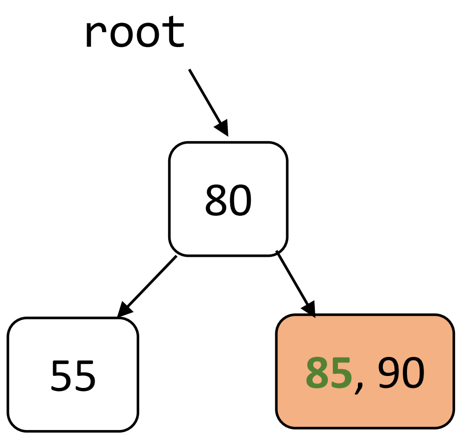


Red-Black tree

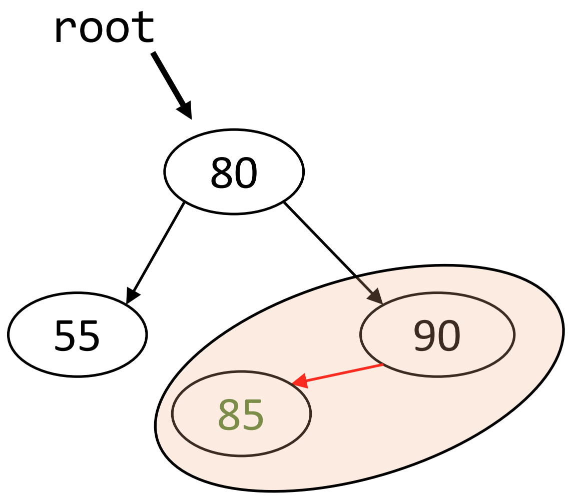


Insertion

2-3 tree



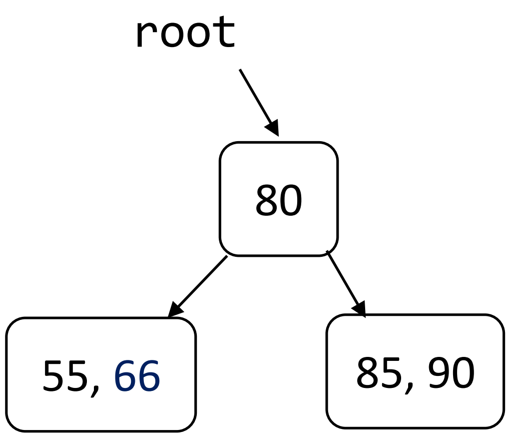
Red-Black tree



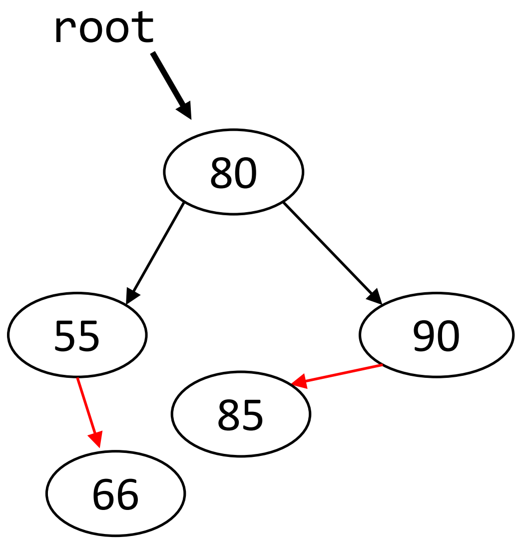
3-node! That's normal! ☺

Insertion

2-3 tree

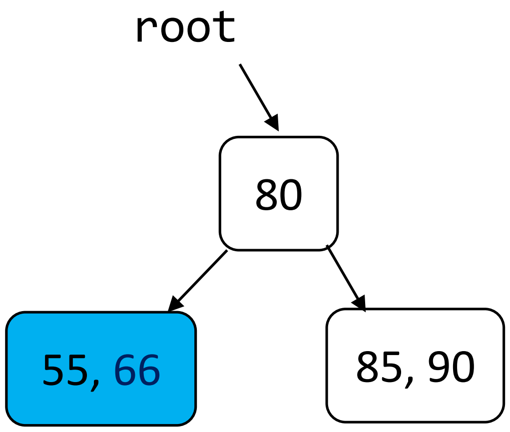


Red-Black tree

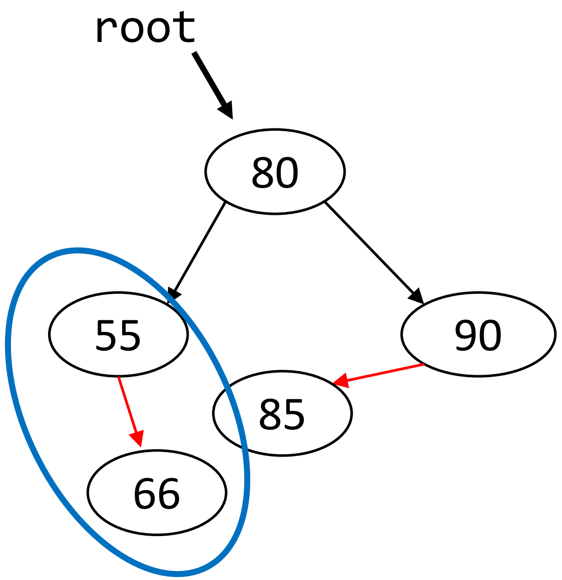


Insertion

2-3 tree

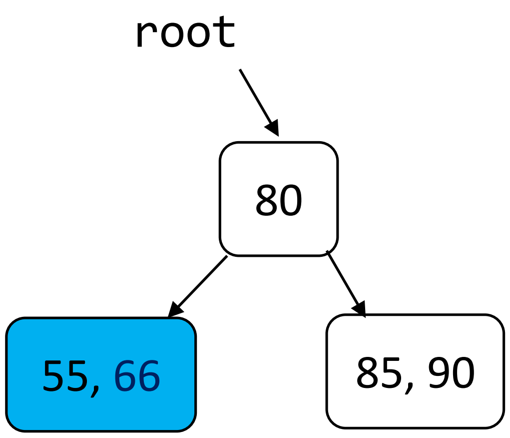


Red-Black tree

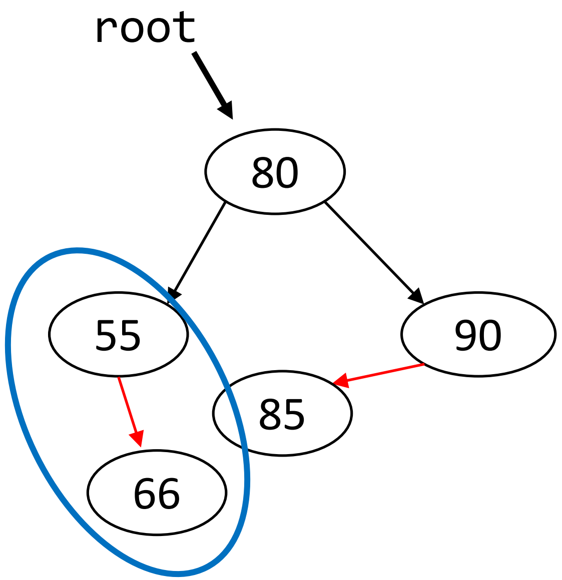


Insertion

2-3 tree



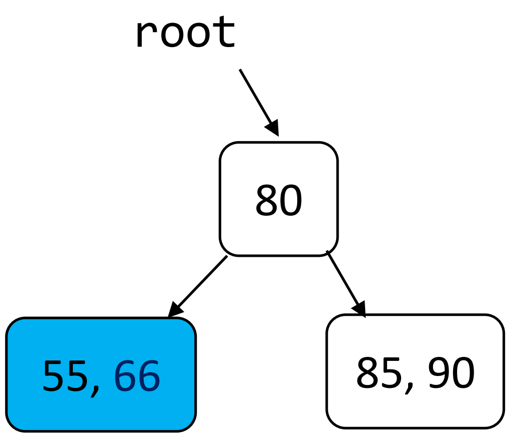
Red-Black tree



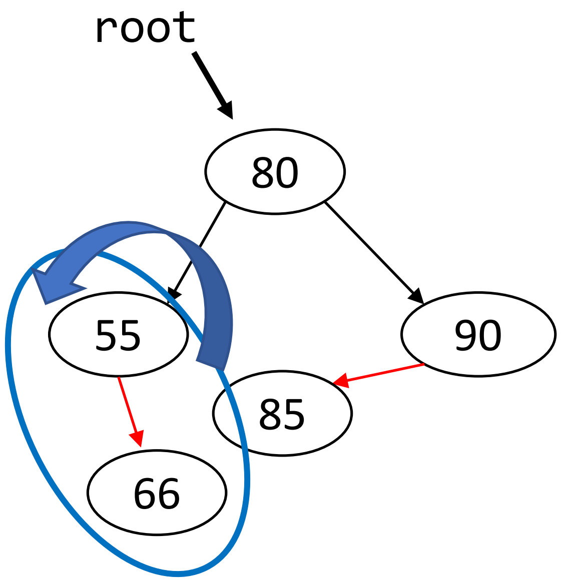
Gotta rotate!

Insertion

2-3 tree



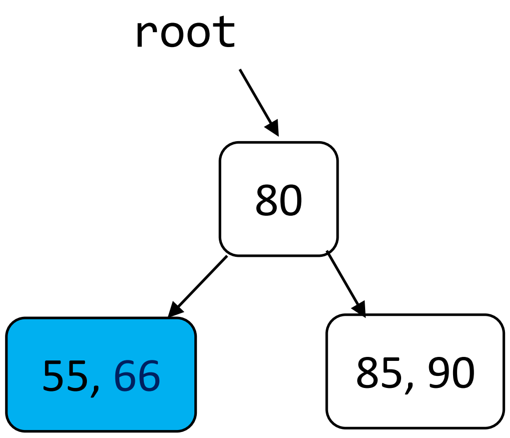
Red-Black tree



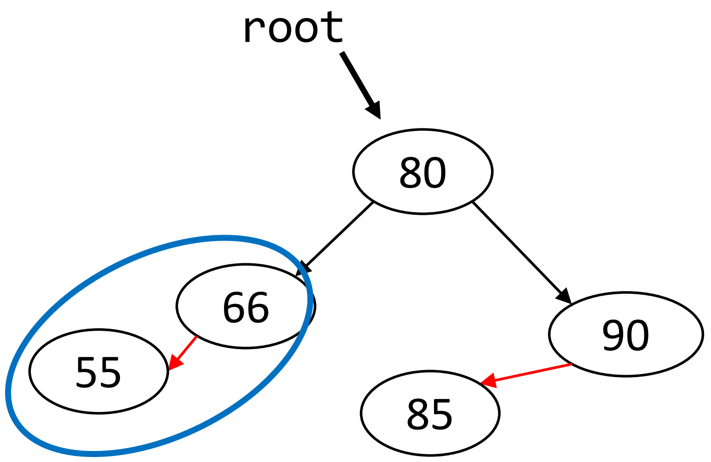
Gotta
rotate!

Insertion

2-3 tree

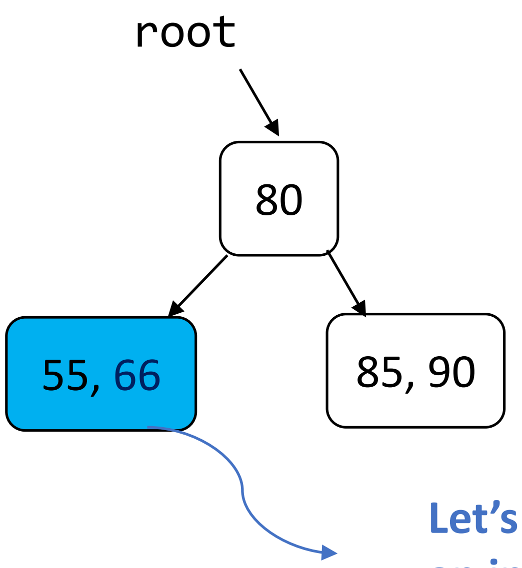


Red-**Black** tree

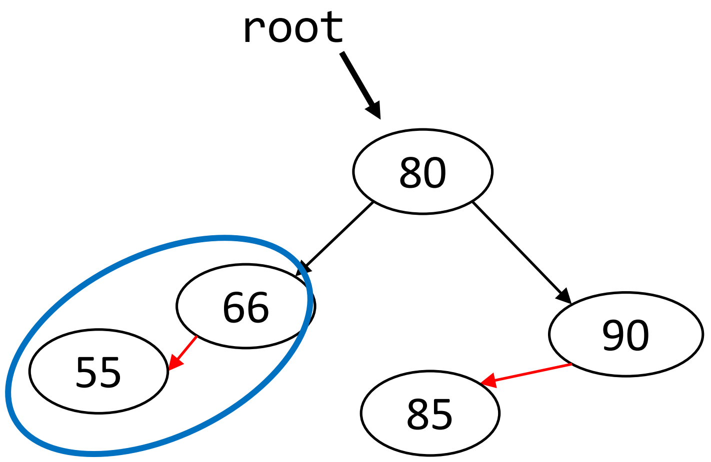


Insertion

2-3 tree



Red-Black tree

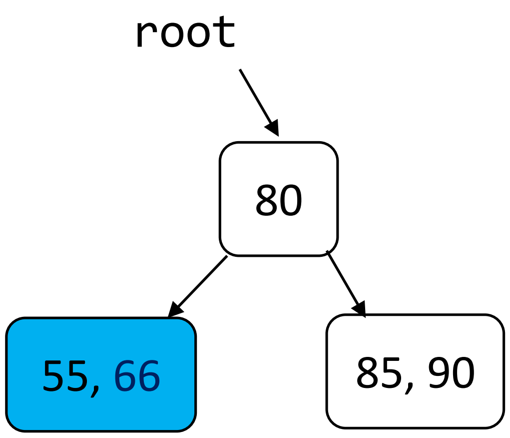


Let's examine three cases for an insertion on this 3-node!

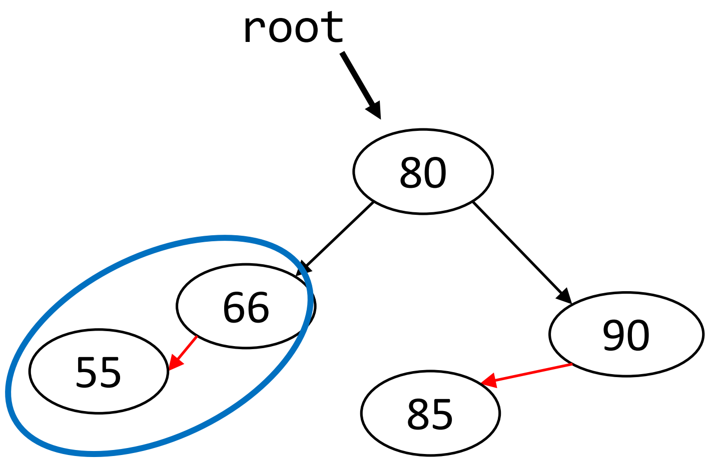
Insertion

2-3 tree

Insert 70



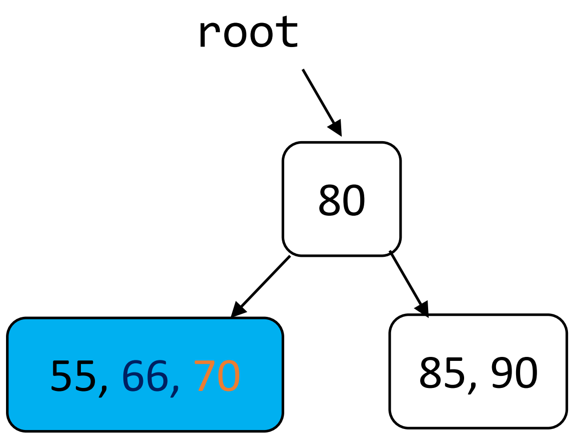
Red-Black tree



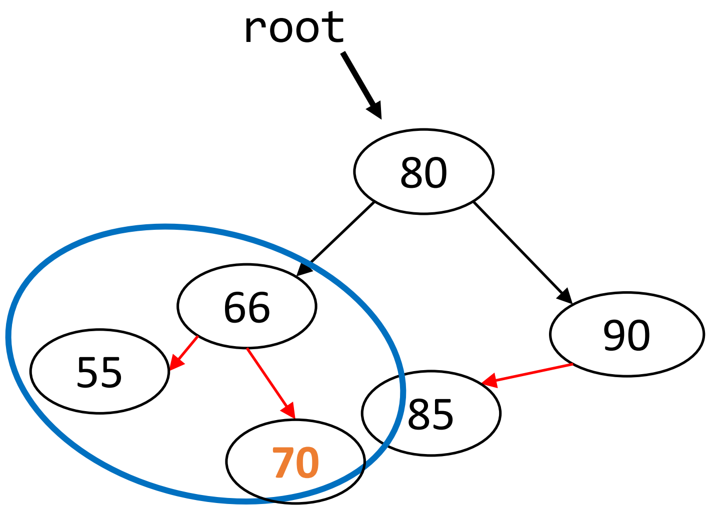
Insertion

2-3 tree

Insert 70



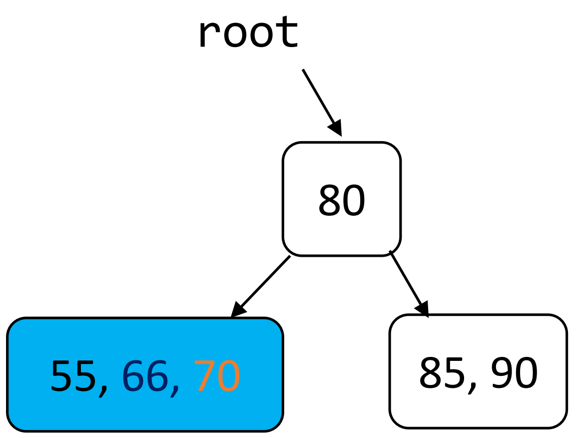
Red-**Black** tree



Insertion

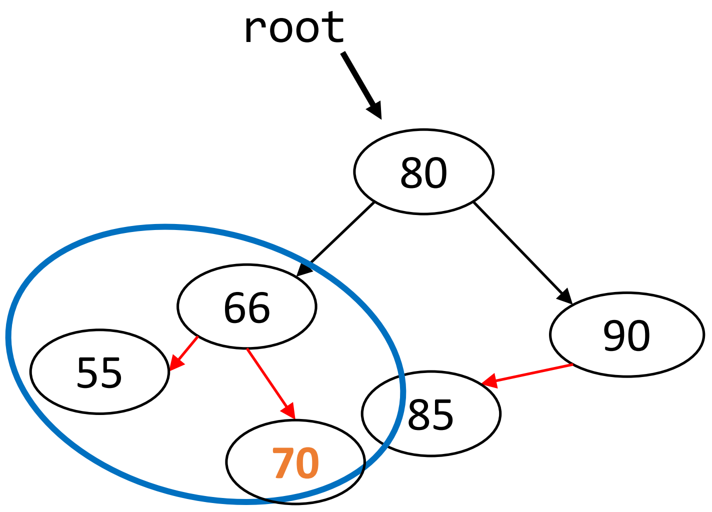
2-3 tree

Insert 70



Overflow! ☹

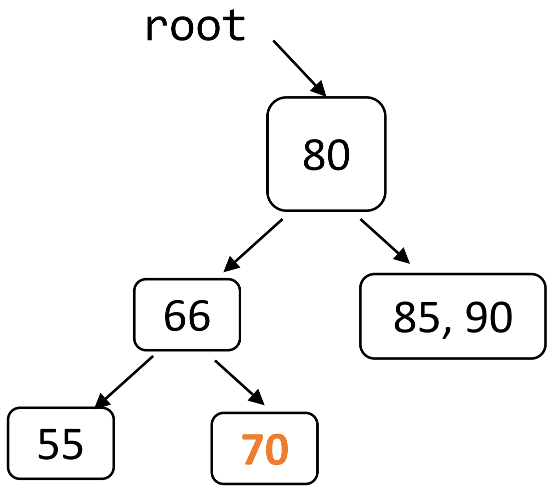
Red-Black tree



`flipColors()!` ☺

Insertion

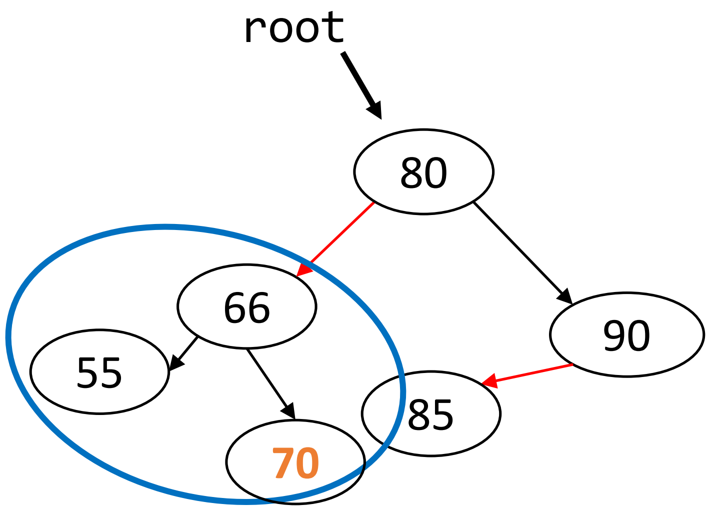
2-3 tree



Insert 70

Overflow

Red-Black tree

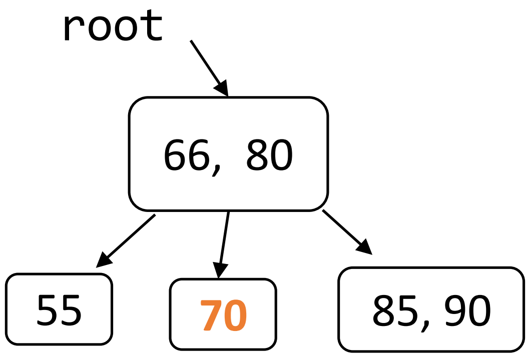


`flipColors()!`

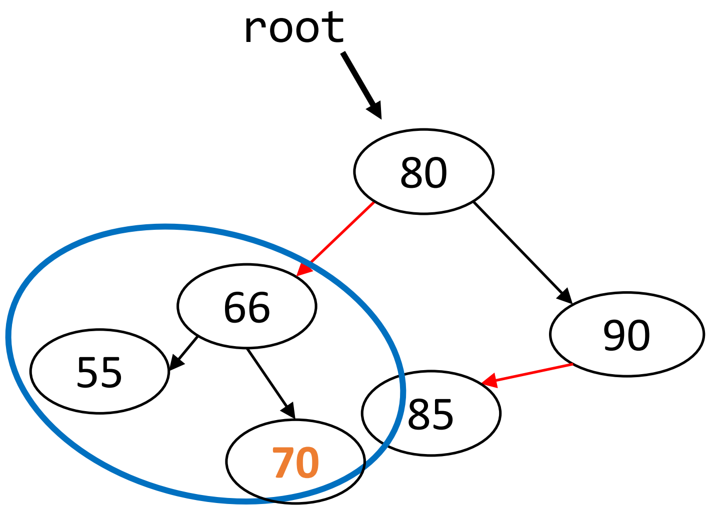
Insertion

2-3 tree

Insert 70



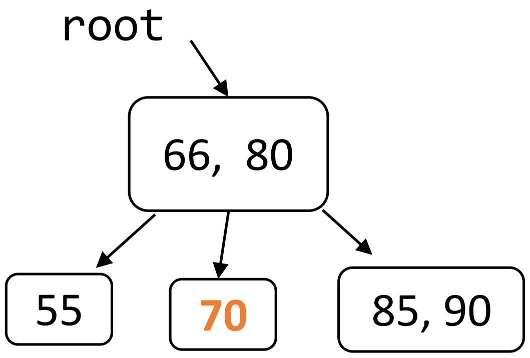
Red-**Black** tree



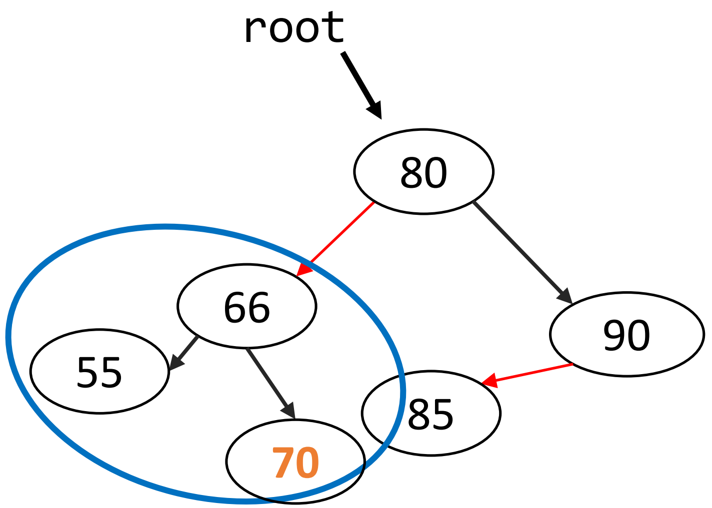
Insertion

2-3 tree

Insert 70



Red-**Black** tree

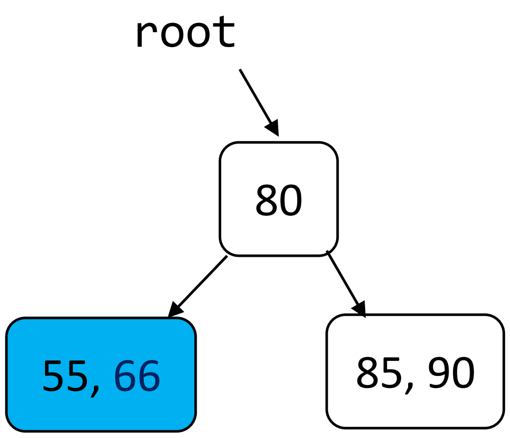


- So, this case, where we insert a key as the biggest key of a temporary 4-node, was rectified with one call to `flipColors()`.
- Notice the perfect **black link** balance of the **Red**-Black Tree?

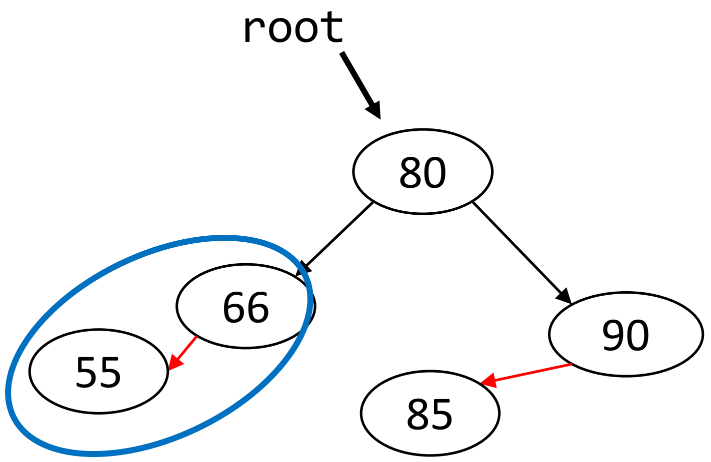
Insertion

2-3 tree

Insert 40



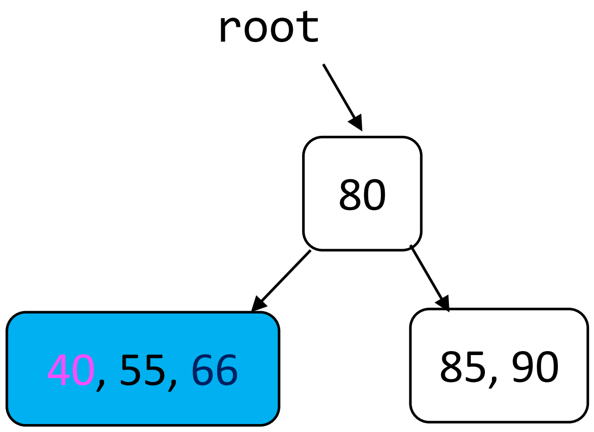
Red-**Black** tree



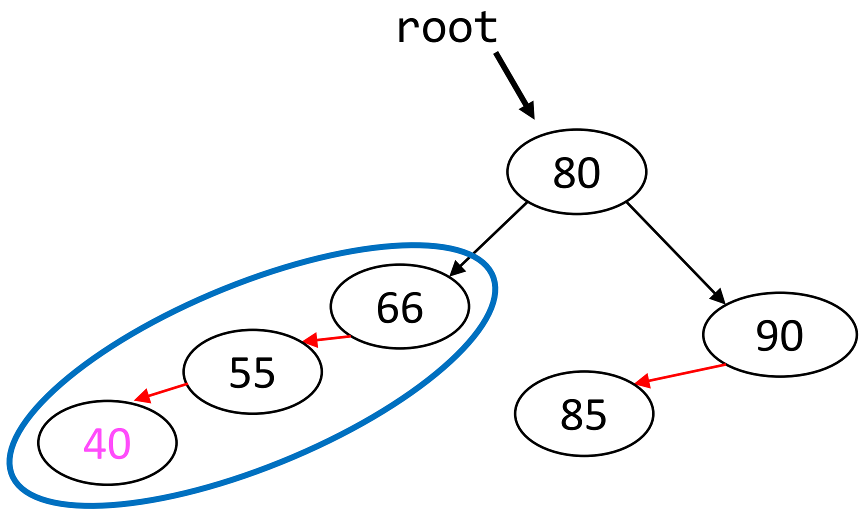
Insertion

2-3 tree

Insert 40



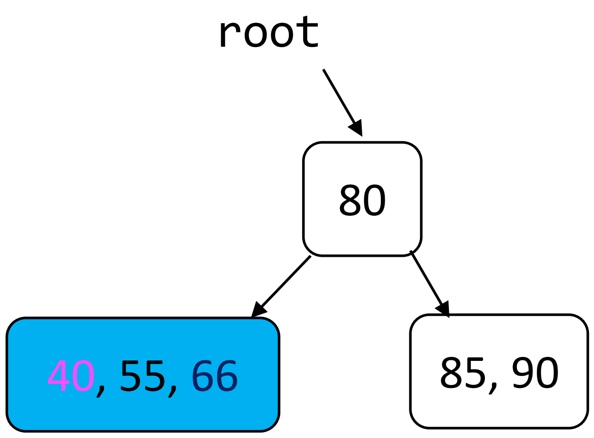
Red-**Black** tree



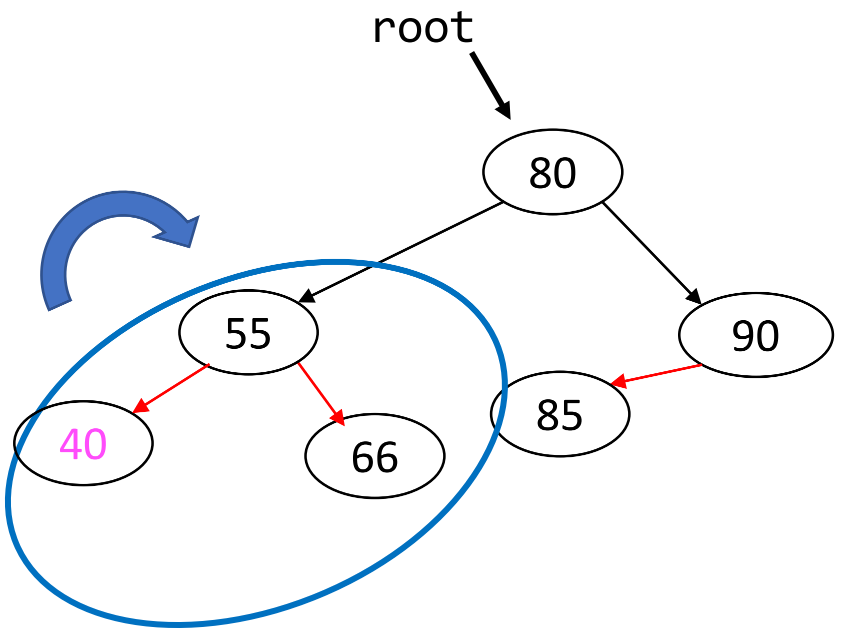
Insertion

2-3 tree

Insert 40



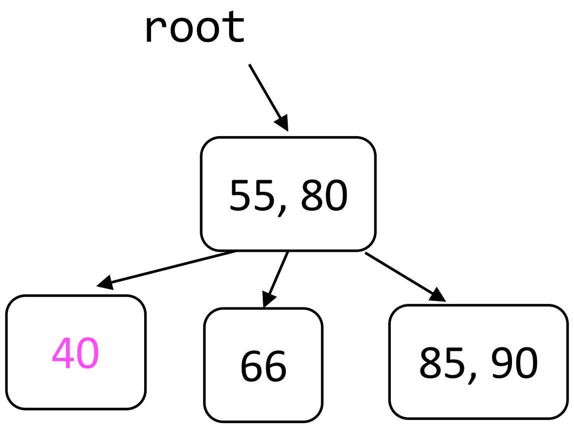
Red-**Black** tree



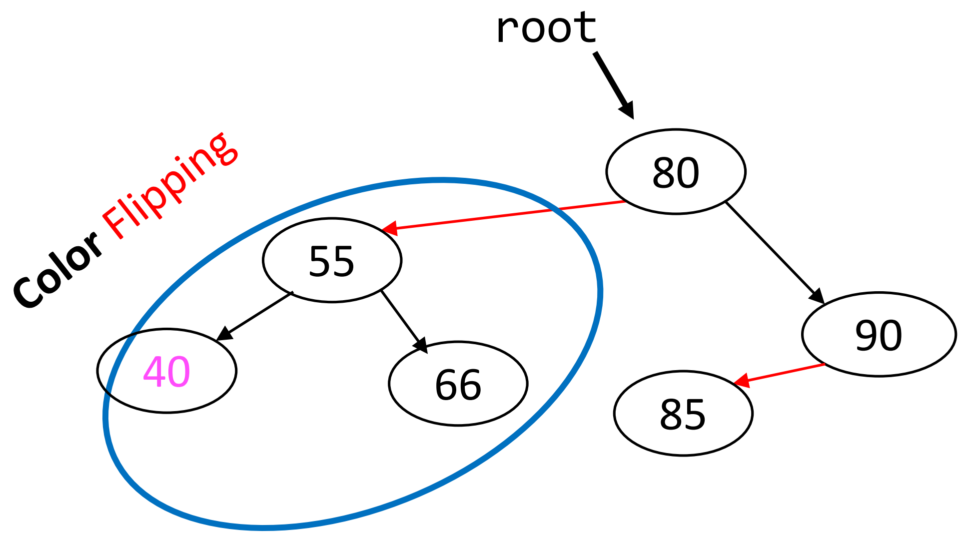
Insertion

2-3 tree

Insert 40

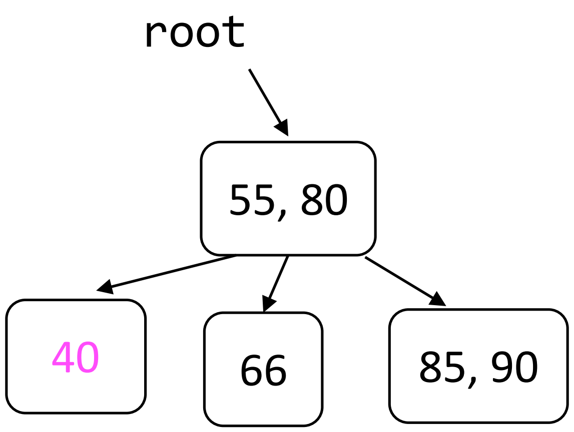


Red-Black tree



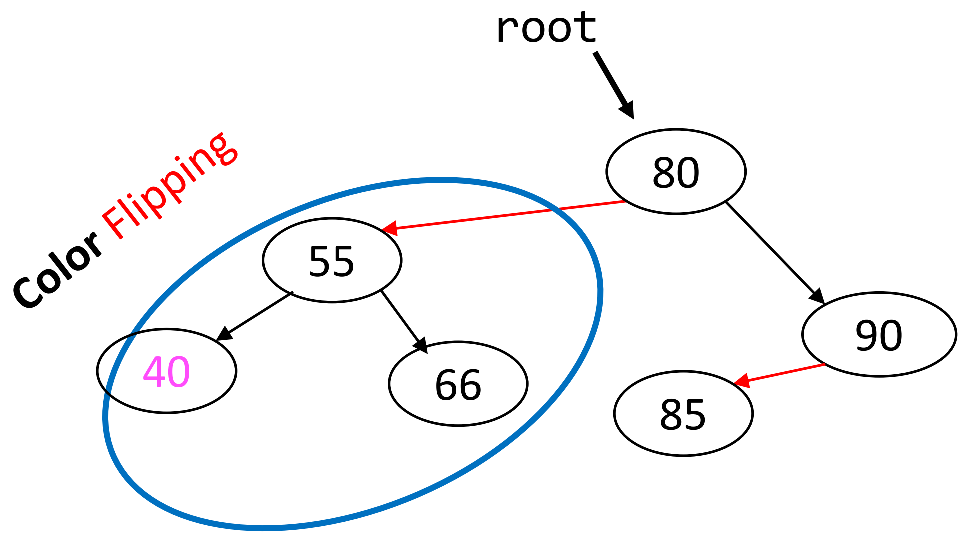
Insertion

2-3 tree



Insert 40

Red-**Black** tree

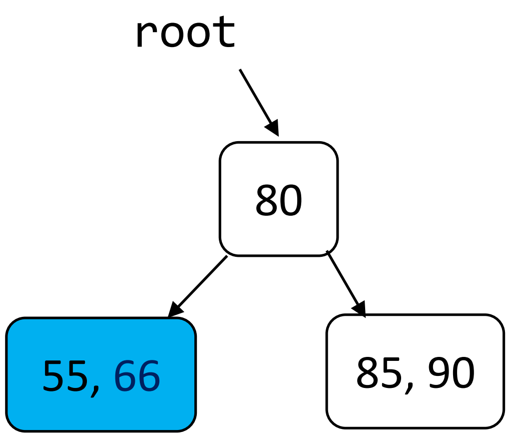


So in this case (inserting as a temporary 4 node's least key), we need 1 rotation and one **color flipping!**

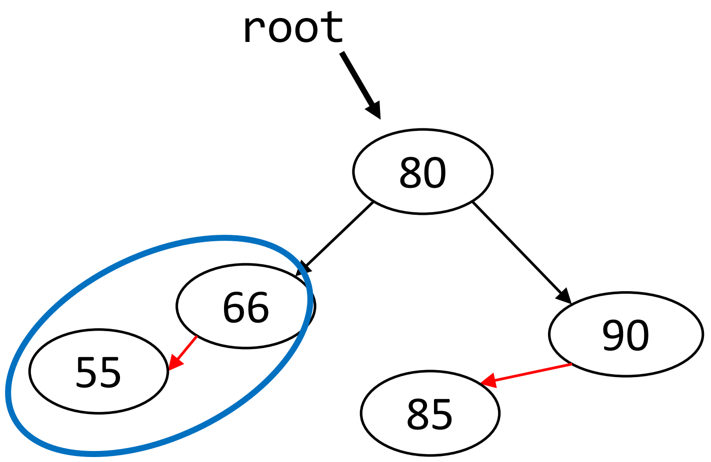
Insertion

2-3 tree

Insert 60



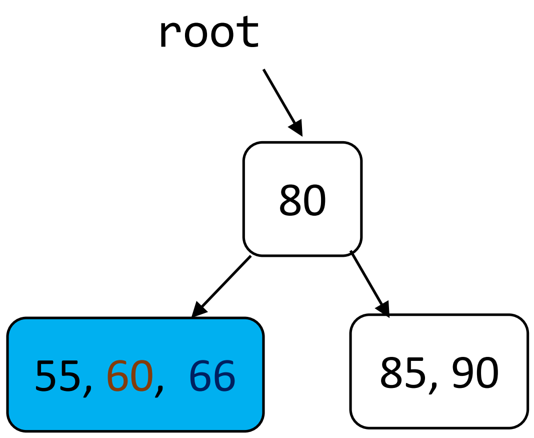
Red-**Black** tree



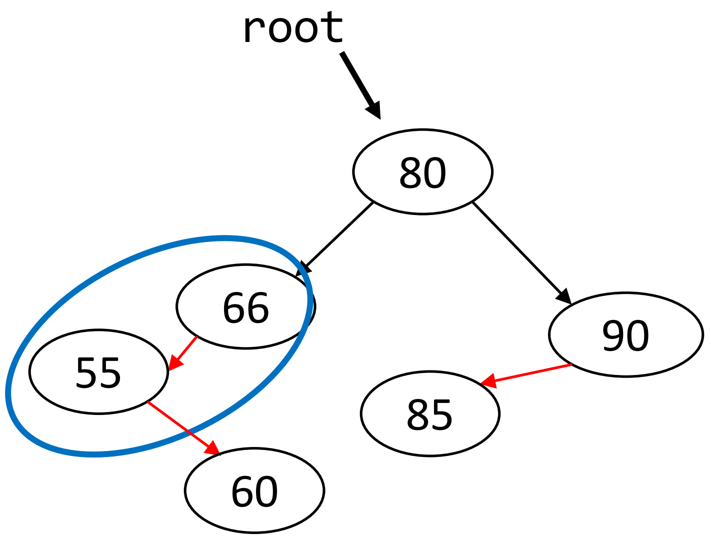
Insertion

2-3 tree

Insert 60



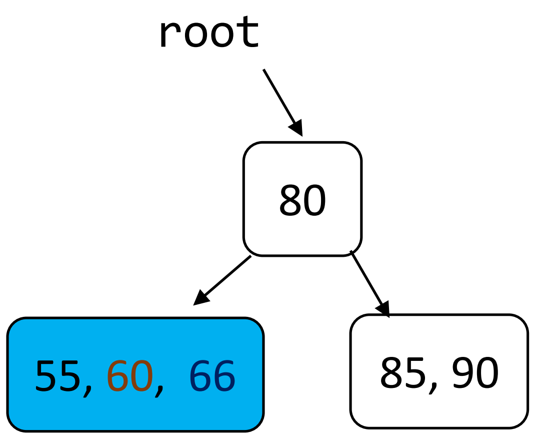
Red-Black tree



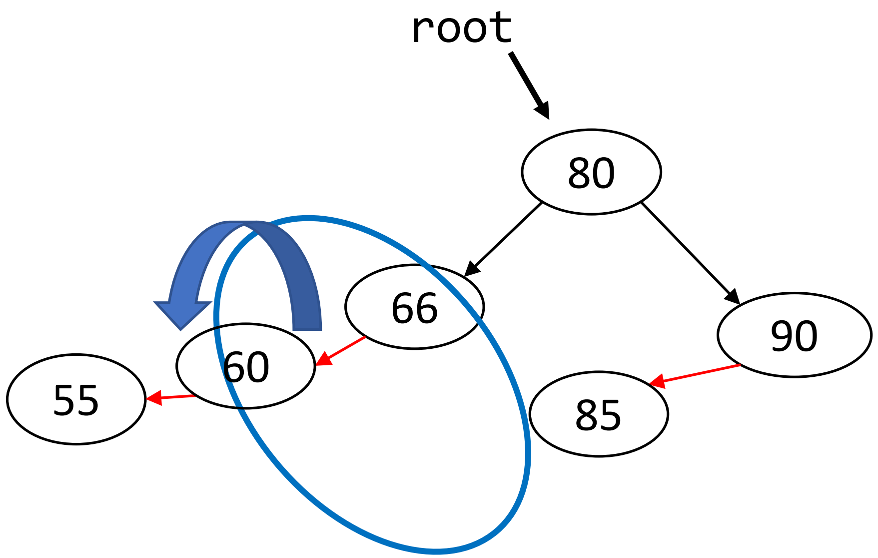
Insertion

2-3 tree

Insert 60

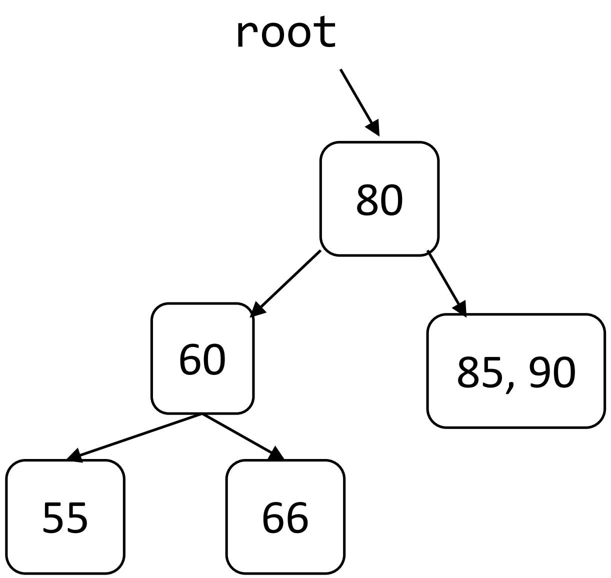


Red-Black tree



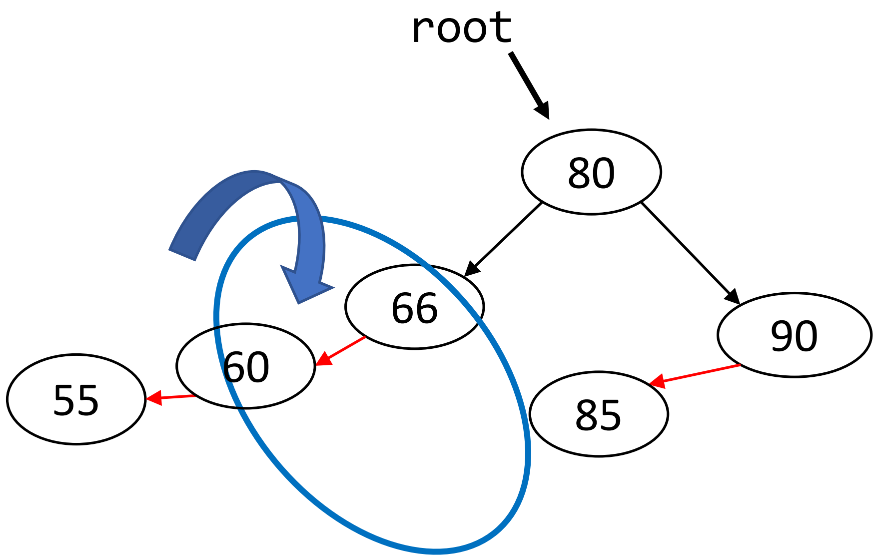
Insertion

2-3 tree



Insert 60

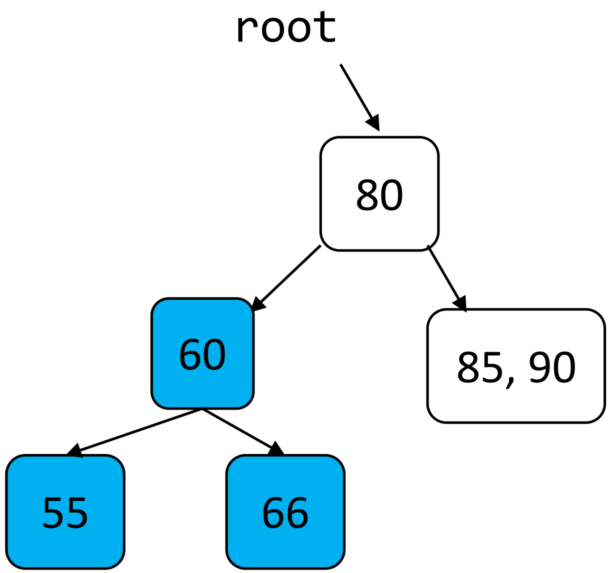
Red-Black tree



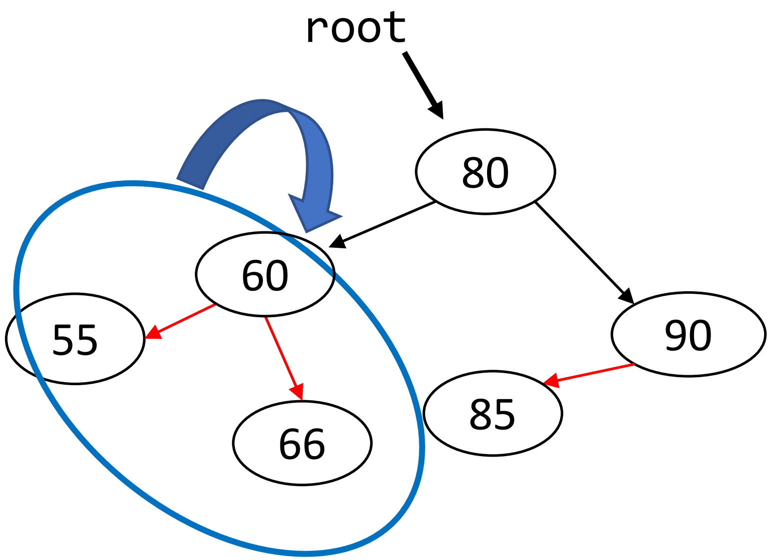
Insertion

2-3 tree

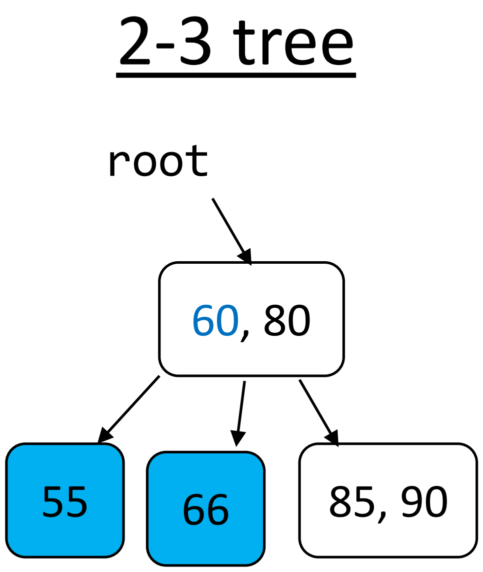
Insert 60



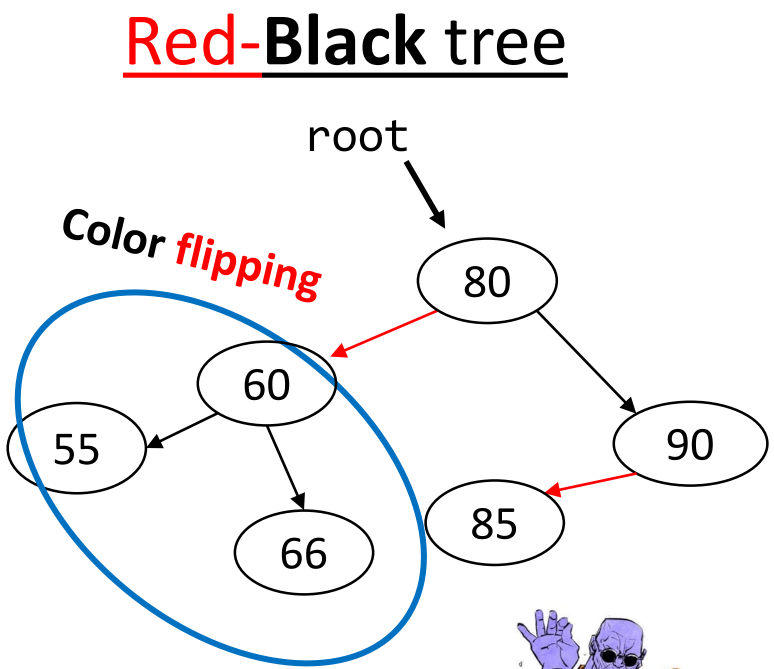
Red-Black tree



Insertion



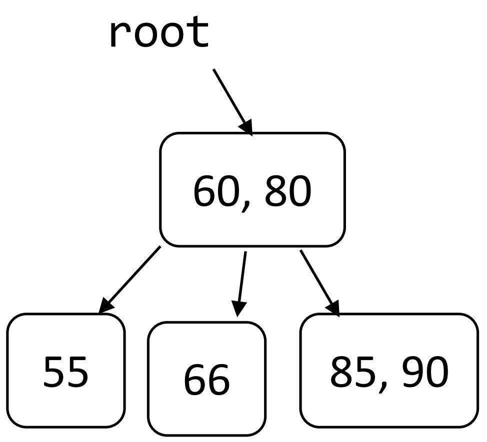
Insert 60



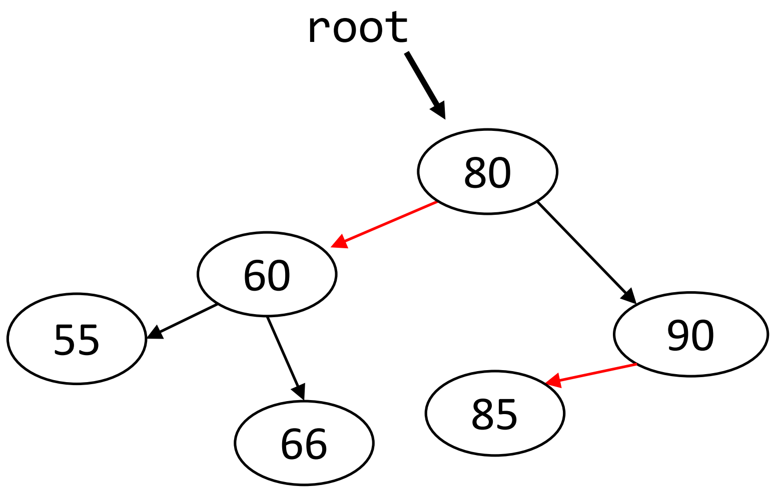
In this final case (inserting as a temporary 4 node's **middle** key), we need 2 rotations and one **color flipping!**

Exercise for you!

2-3 tree



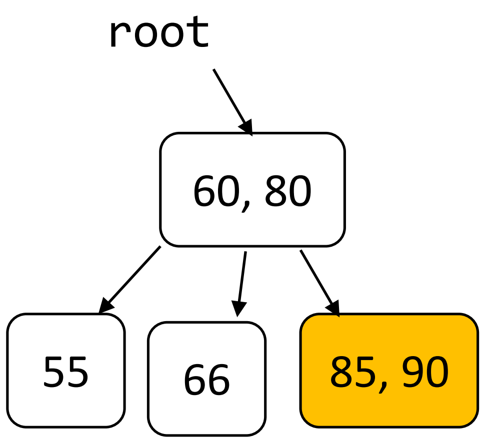
Red-**Black** tree



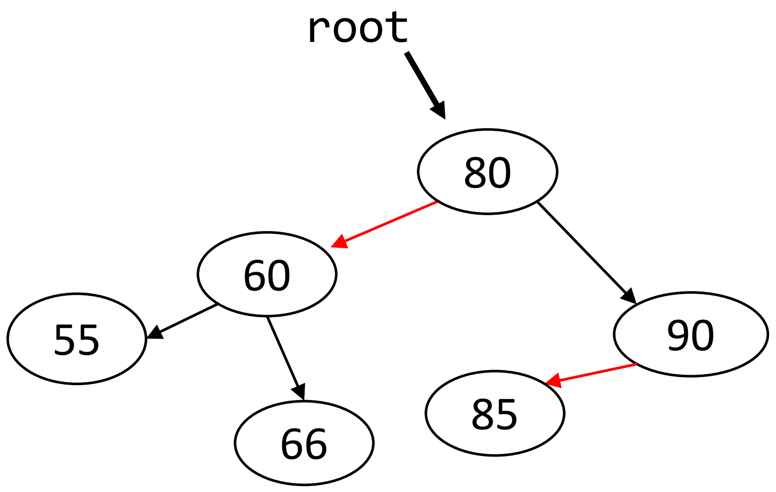
Please go ahead and insert 87 for me please! ☺

Exercise for you!

2-3 tree



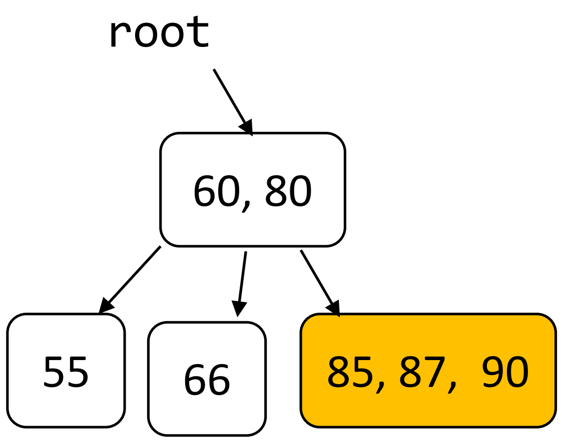
Red-**Black** tree



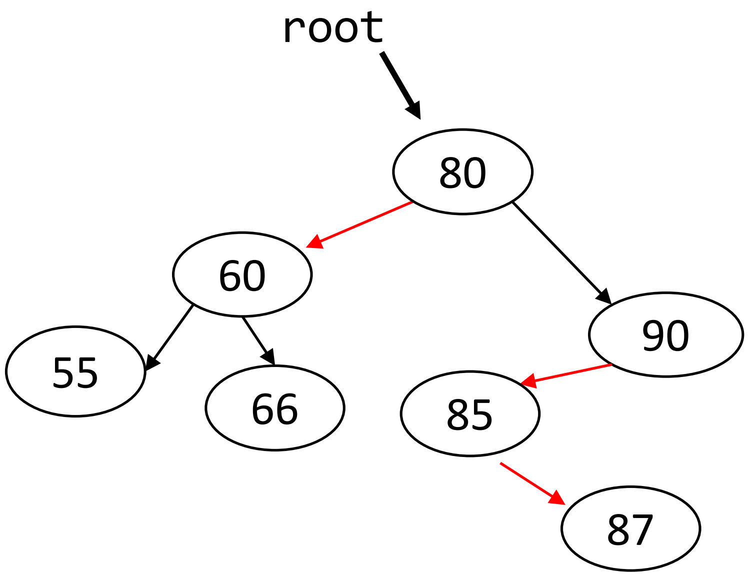
Remember: In Red-Black trees, we can't model key rotations! We will split the yellow node!

Exercise for you!

2-3 tree

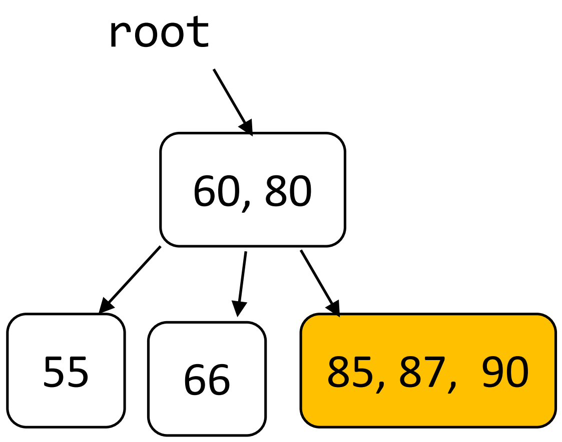


Red-**Black** tree

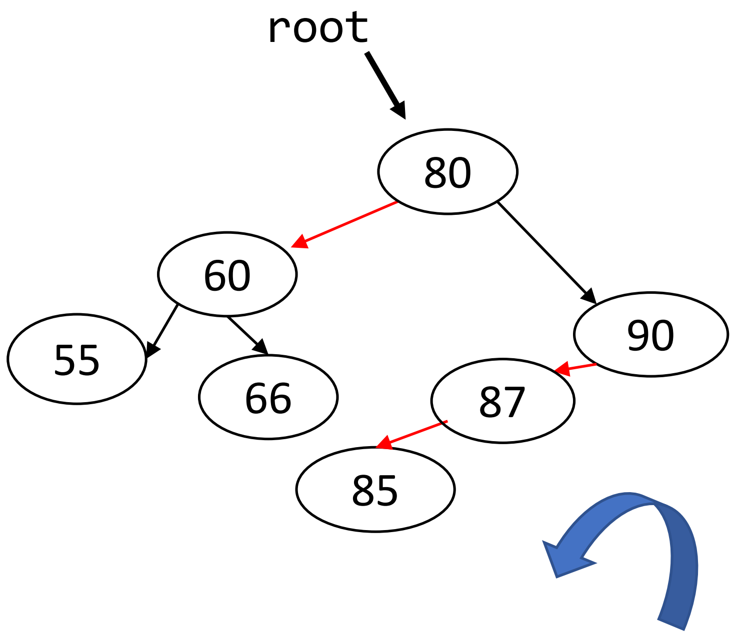


Exercise for you!

2-3 tree

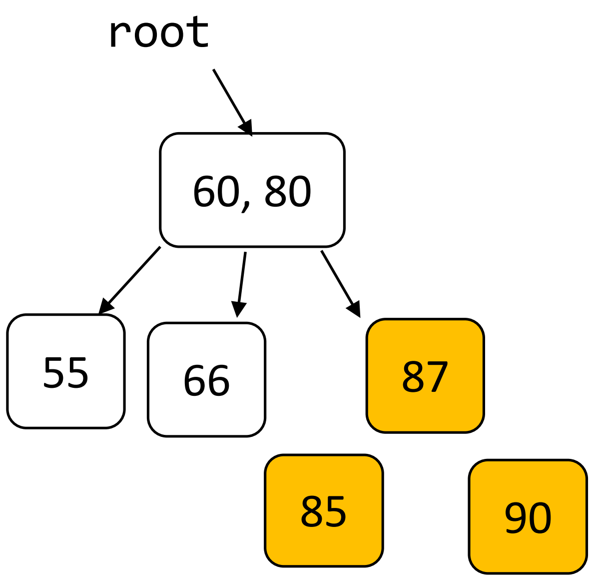


Red-**Black** tree

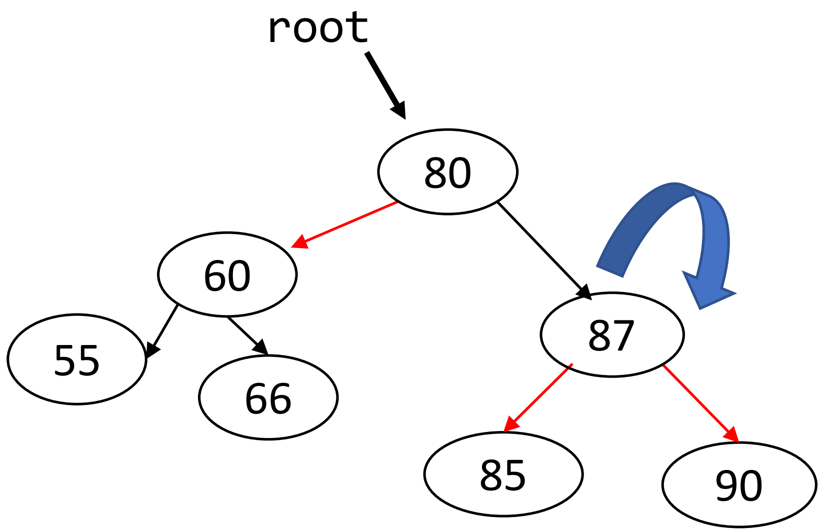


Exercise for you!

2-3 tree

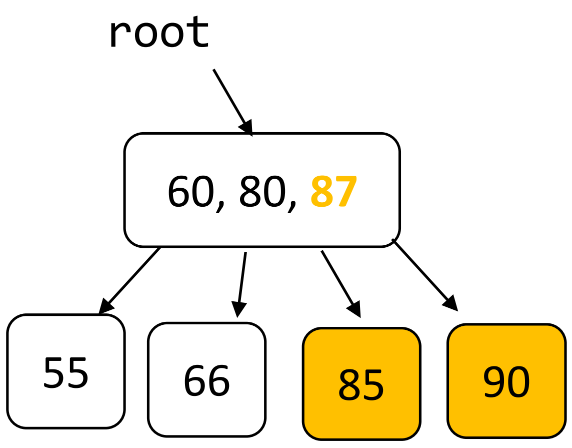


Red-Black tree

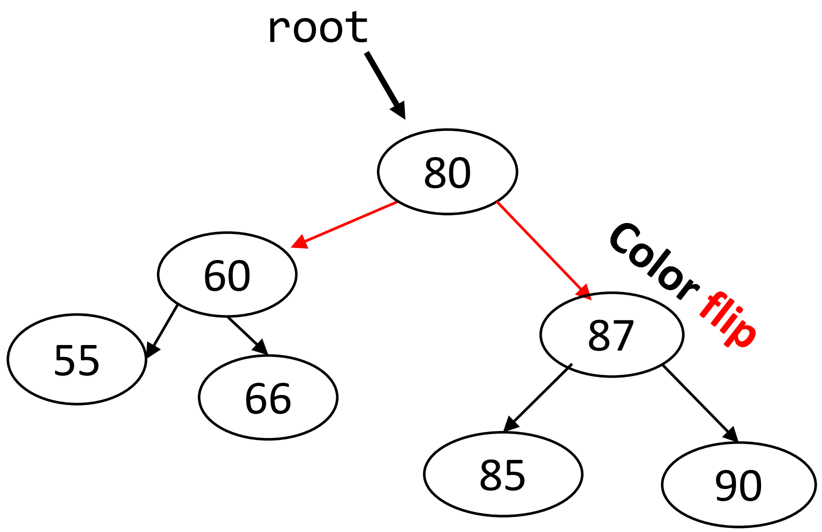


Exercise for you!

2-3 tree

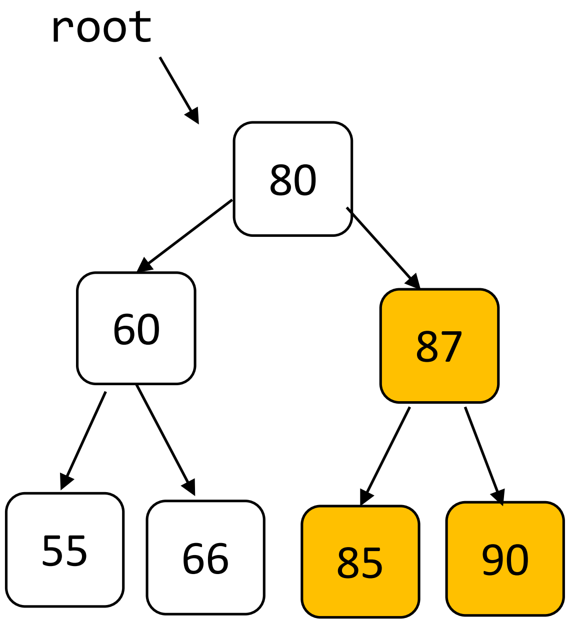


Red-Black tree

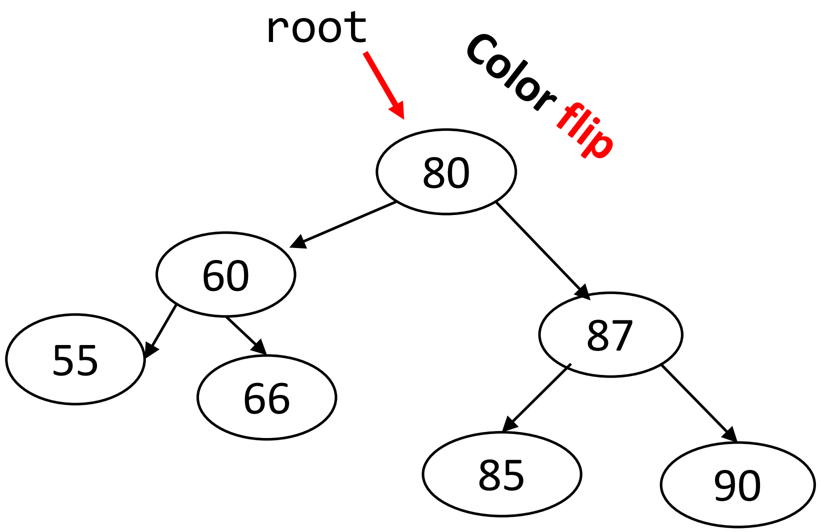


Exercise for you!

2-3 tree

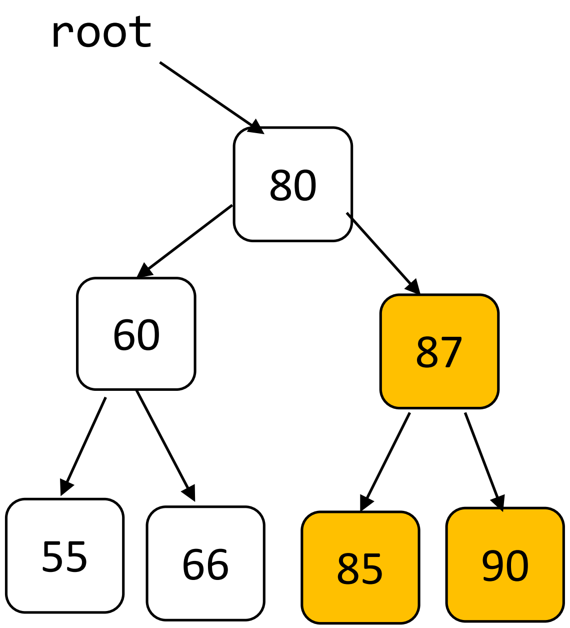


Red-**Black** tree



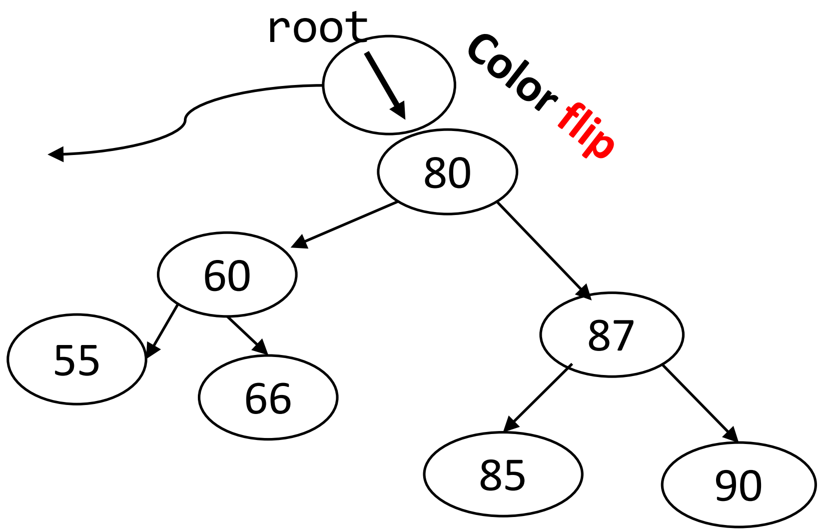
Exercise for you!

2-3 tree



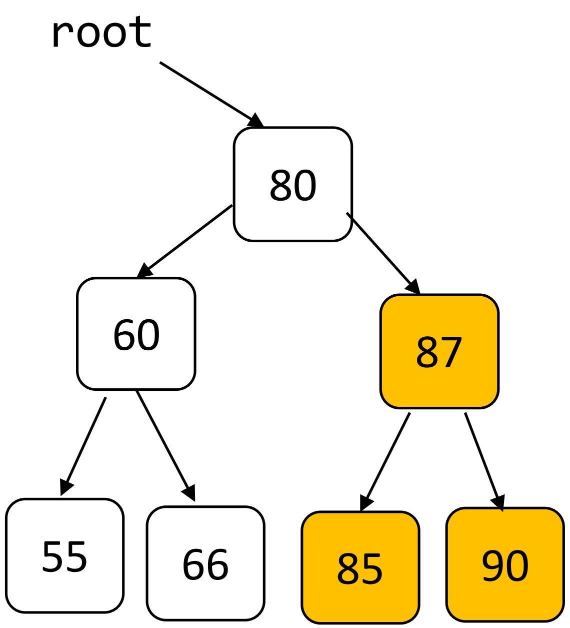
Red-Black tree

Don’t forget to set the root to **BLACK!**

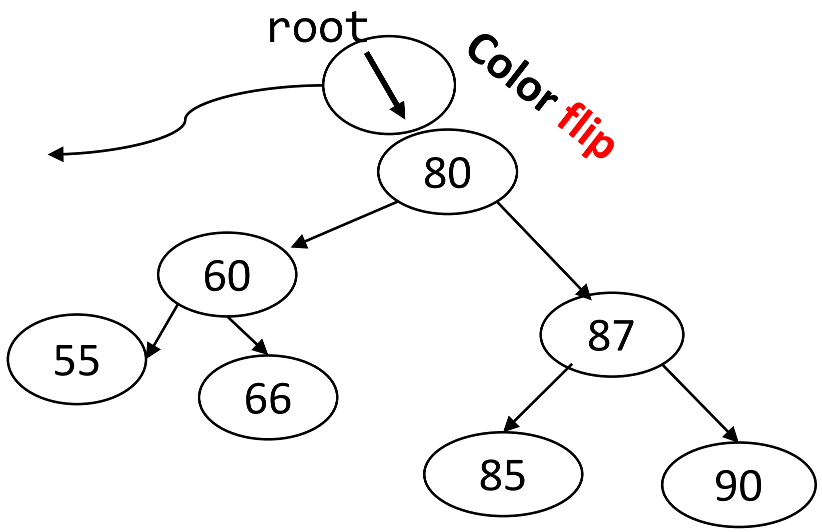


Exercise for you!

2-3 tree



Red-Black tree



Don't forget to set the root to **BLACK!**

```
public void insert(Key key){
    root = insert(root, key);
    root.color = BLACK;
}
```

Speaking of...

Fill-in the insertion subroutine!

```
private Node insert(Node n, Key key){
    if(n == null) return new Node(key, RED);
    if(key.compareTo(n.key) < 0)
        n.left = insert(n.left, key);
    else if(key.compareTo(n.key) >= 0)
        n. right = insert(n.right, key);
    else return n;
    if(isRed(n.right) && __________ ) n = rotateLeft(n);
    if(isRed(n.left) && __________) n = rotateRight(n);
    if(_______ && isRed(n.right)) __________;

}
```

```
boolean isRed(Node n){
  if(n == null) return false;
  else return (n.color == RED);
```

Fill-in the insertion subroutine!

```
private Node insert(Node n, Key key){
    if(n == null) return new Node(key, RED);
    if(key.compareTo(n.key) < 0)
        n.left = insert(n.left, key);
    else if(key.compareTo(n.key) >= 0)
        n. right = insert(n.right, key);
    else return n;
    if(isRed(n.right) && _!isRed(n.left)___ ) n = rotateLeft(n);
    if(isRed(n.left) && __isRed(n.left.left)____) n = rotateRight(n);
    if(__isRed(n.left)_____ && isRed(n.right)) __flipColors(n)__;

}
```

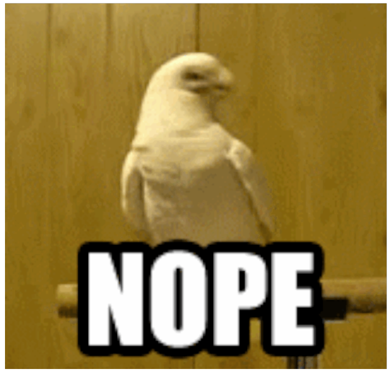
```
boolean isRed(Node n){
  if(n == null) return false;
  else return (n.color == RED);
```

Problem 3!

“Hard” Deletion

- By “hard” Deletion we refer to a process that actually removes the key’s wrapper, the node, from the tree.
  - We’ve already seen how to do this in threaded and non-threaded BSTs, AVL, Splay, 2-3 (conceptually).

“Hard” Deletion



- By “hard” Deletion we refer to a process that actually removes the key’s wrapper, the node, from the tree.
  - We’ve already seen how to do this in threaded and non-threaded BSTs, AVL, Splay, 2-3 (conceptually).
- Hard deletion in RBBSTs is way too hard and we will not cover it.
  - It relies on subroutines for deleting the minimum and maximum of a subtree.
  - Segdewick and Wayne show an implementation as an exercise in 3.3.

“Soft” deletion (Mark – and – Sweep)

- Instead, in practice, we perform a Mark-and-Sweep step.
- A key deletion does not lead to immediate freeing of the key’s node and re-placement of links.
  - Instead, a bit is set to indicate that the relevant object can be safely garbage – collected
  - Note that since it can be garbage-collected, it can also be replaced!
    - As long as **we are sure** we can place a node with our new key exactly where the old node used to be…
- Every T ms, a “sweeping” phase runs, collecting only the objects with unset bits.
  - Those objects are re-inserted in another tree and the root reference is assigned to that new tree so the old one can be garbage collected.

Some theory behind RBBSTs

- Red-Black Trees have perfect ***black link balance***
  - All paths from the root to a null pointer dereference the same # black pointers!
  - So, if we somehow could ignore red links (we really shouldn't, we'd miss keys), then we'd have excellent search all the time!
- Remember: in classic BSTs, search can be as bad as $\mathcal{O}(n)$, for n keys and unit cost that of a pointer dereferencing.
- In RBBSTs, you are guaranteed $\mathcal{O}(\log_2 n)$ irrespective of insertion order.

Some theory behind RBBSTs

- Red-Black Trees have perfect ***black link balance***
  - All paths from the root to a null pointer dereference the same # black pointers!
  - So, if we somehow could ignore red links (we really shouldn't, we'd miss keys), then we'd have excellent search all the time!
- Remember: in classic BSTs, search can be as bad as $\mathcal{O}(n)$, for n keys and unit cost that of a pointer dereferencing.
- In RBBSTs, you are guaranteed $\mathcal{O}(\log_2 n)$ irrespective of insertion order.

Some theory behind RBBSTs

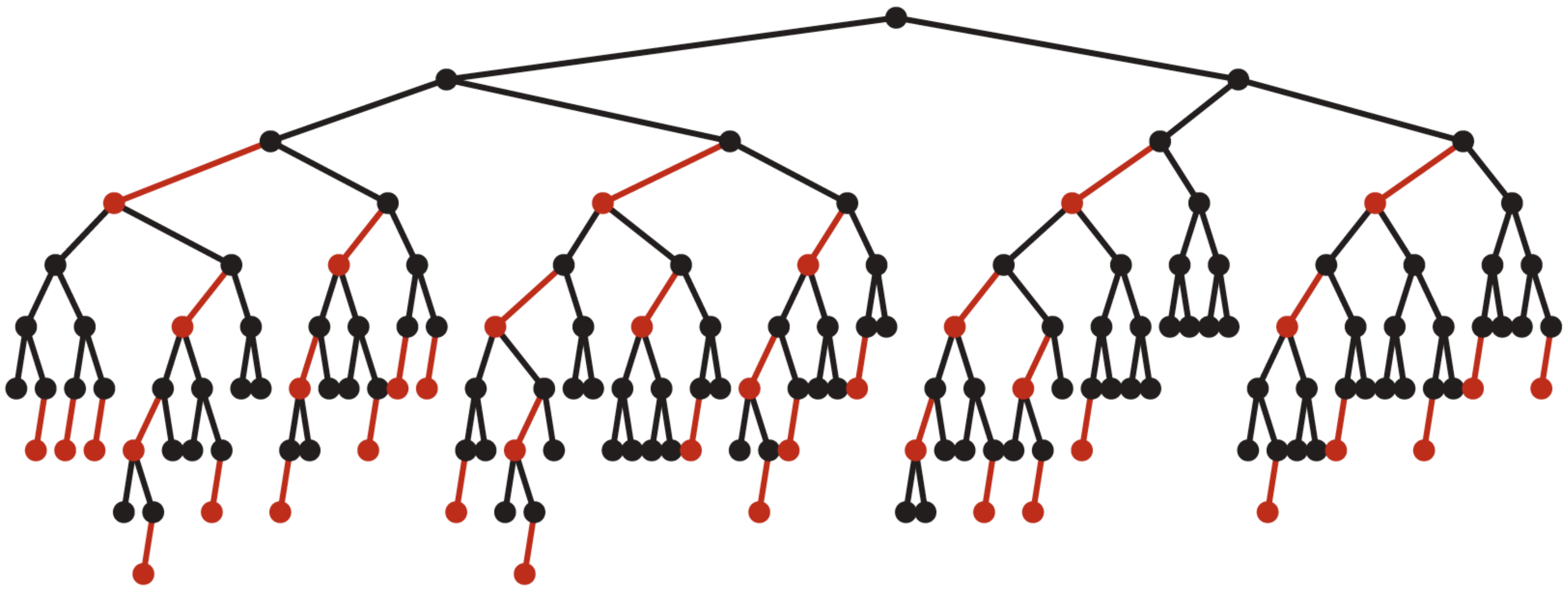
- Red-Black Trees have perfect ***black link balance***
  - All paths from the root to a null pointer dereference the same # black pointers!
  - So, if we somehow could ignore red links (we really shouldn't, we'd miss keys), then we'd have excellent search all the time!
- Remember: in classic BSTs, search can be as bad as $\mathcal{O}(n)$, for n keys and unit cost that of a pointer dereferencing.
- In RBBSTs, you are guaranteed $\mathcal{O}(\log_2 n)$ irrespective of insertion order.
  - *DIDN'T WE ALSO HAVE THIS IN AVL TREES?*

Some theory behind RBBSTs

- It turns out that it is only rarely that RBBSTs will have anything close to $2\log_2 n$ height!
- To minimize the height, we need to minimize the red links.

Average height in RBBSTs

- Turns out that in practice, RBBSTs look more like this…



AVL vs Red-Black

AVL		**Red-Black**	
+	-	+	-
Height $\mathcal{O}(\log_2 n)$	Spatial overhead	Height $\mathcal{O}(\log_2 n)$	Height can be greater than $\lfloor \log_2 n \rfloor$
Easier (hard) deletions		Can help us implement both 2-3 and 2-3-4 trees	Hard deletion hard to implement