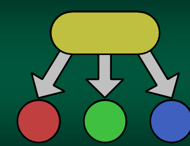
Balanced Trees

Section 3.3

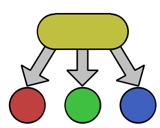
2-3 Trees



Balance using really big nodes

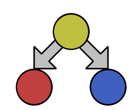
2-3 Trees

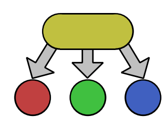
- The *2-3 Tree* is a special type of BST invented by *John Hopcroft* in 1970
- *It automatically maintains balance as it grows!*
- It does this by using a clever variation of the node that can contain multiple values



2-3 Trees

- *2-Nodes* contain 1 values and two children: left and right
- *3-Nodes* contains 2 values and three children: left, middle and right
- Both are easily to code and traversal logic is straight forward





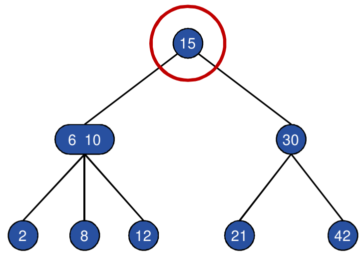
Searching a 2-3 tree

- Searching a 2-3 Tree is very similar to a Binary Search Tree, but with a minor difference
- 2-nodes are treated the same as they are in BSTs:
  - if less than, go left
  - if greater than, go right

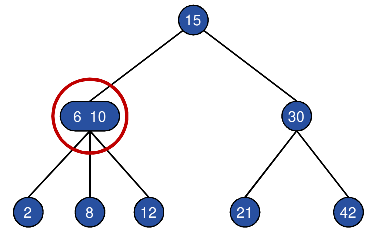
Searching a 2-3 tree

- 3-nodes are a bit different
- Since they have 2 values, *a* and *b*, we do the following:
  - if less than *a*, go left
  - if between *a* and *b*, go middle
  - if greater than *b*, go right

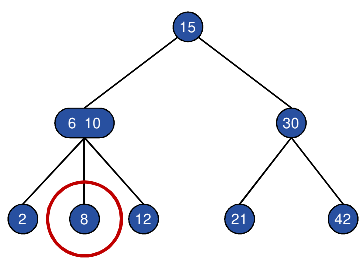
Search for 8: Go Left



Search for 8: Go Middle

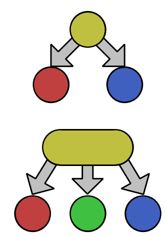


Search for 8: Found



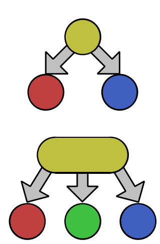
Adding to a 2-3 tree

- For BSTs, when a value is added, it will search and then create a new left or right leaf
- 2-3 Trees, however, will merge the value into the bottom node (rather than creating a new node)

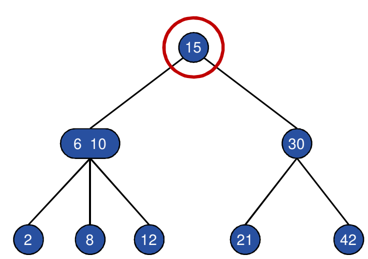


Adding to a 2-3 tree

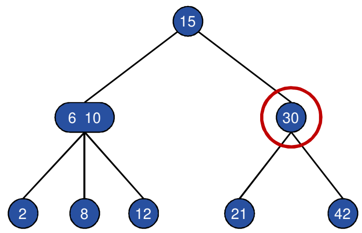
- This will convert a 2-Node into a 3-Node (it now has two values and three links)
- A 3-Node will convert into a temporary structure called a 4-Node… but we will get to that later



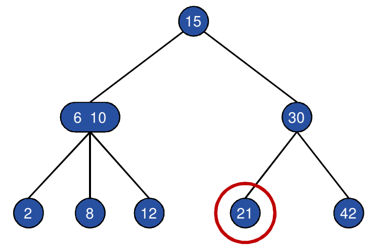
Add 25: Go Right



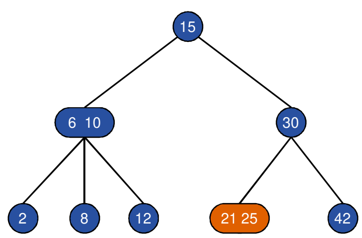
Add 25: Go Left



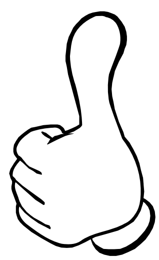
Add 25: Can't go further



Add 25: Convert 2-Node to 3-Node



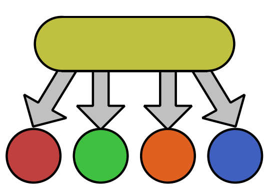
Adding to a 2-3 tree



- Notice, when the value was added to the 2-3 Tree, that the height of the tree *did not change*
- Binary Search Tree would have added another child node and the height *would have changed*

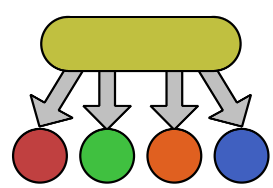
The 4-Node

- So, what happens when we add a value to a 3-node?
- It becomes a *4-Node*, which has 3 values and 4 children
- **This is temporary**, it will be converted

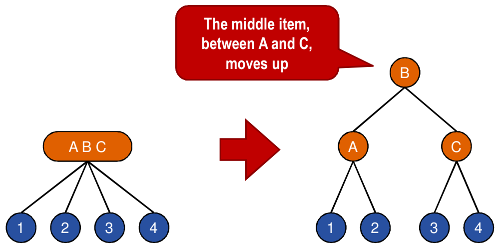


The 4-Node

- When a 4-Node is created, the 2-3 Tree algorithm will *split* it into other nodes
- Given that 4 is a nice even number, we can split equally
- … and balanced!

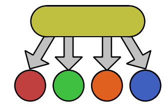


One Way to Split a 4-Node



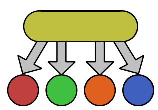
The Types of Splits

- There are a total of six different splits that can occur in a 2-3 tree
- In each split, the **middle** value **ascends** up to the parent node



The Types of Splits

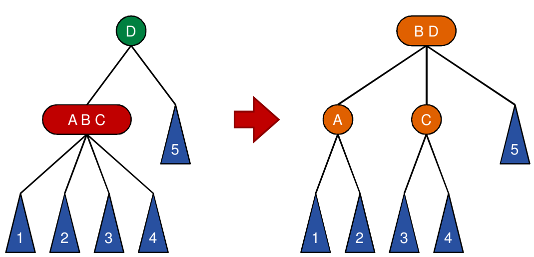
- This will change a parent from a 2-Node to 3-Node
- … or from 3-Node to 4-Node
  - then, the parent will split
  - it continues to bubble up – possibly all the way to the root
  - this is O(log n)



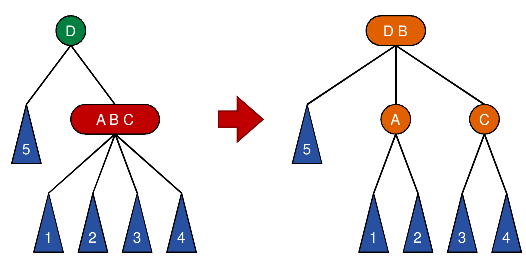
The Six Splits

- Parent is 2-Node:
  - node is the left child of the parent (1)
  - node is the right child of the parent (2)
- Parent is 3-Node:
  - node is the left child of the parent (3)
  - node is the middle child of the parent (4)
  - node is the right child of the parent (5)
- Node is the root (6)

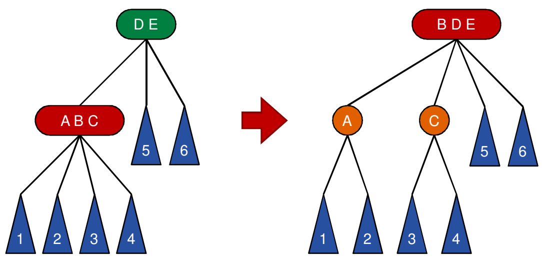
2-Node Parent, Left Child



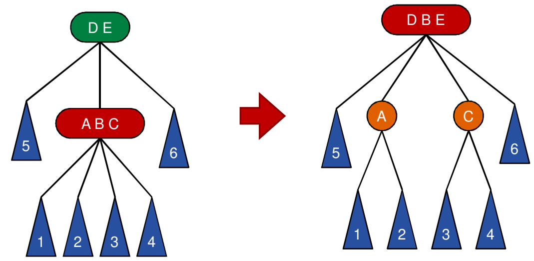
2-Node Parent, Right Child



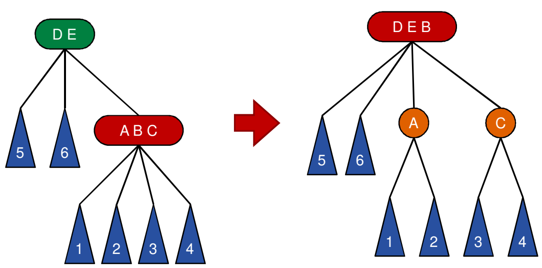
3-Node Parent, Left Child



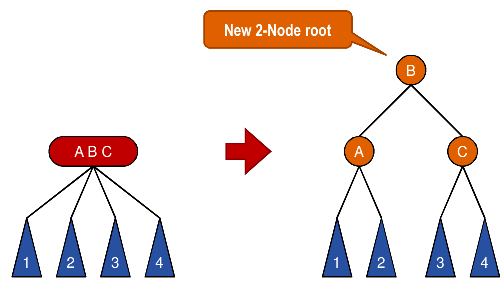
3-Node Parent, Middle Child



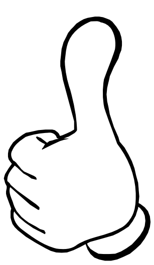
3-Node Parent, Right Child



Node is a the root

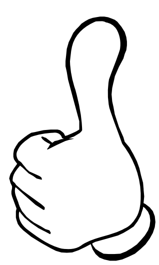


Why Does This Work?



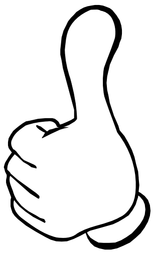
- Notice that, of the six splits, only <u>one</u> created a new node and changed the height
- So, *a 2-3 tree grows in depth only when the root is split*
- … and it splits balanced on the left and right side!

Why Does This Work?



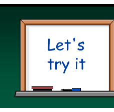
- 2-3 Trees grow from the top rather than from the bottom like Binary Search Trees
- And, the tree auto-balances do to the very nature of how the nodes split
- They are always O(log n)

Why Does This Work?



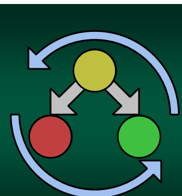
- Additional, 2-3 Trees are *incredibly* easy to write
- When a recursive call completes, in the case of a split, you can return the middle value
- So, as recursion bubbles up, you can handle all splits

Let's Try Some…



- Let's try two sets of numbers inserted into a Binary Search Tree and 2-3 Tree
- This will allow us to contrast the difference

```
1. Numbers: 6, 2, 1, 3, 5, 4, 7, 8
2. Numbers: 1, 2, 3, 4, 5, 6, 7, 8
```

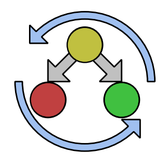
Tree Rotations

Node Sa Node

(if you get this play-on-words, I will be pleasantly surprised)

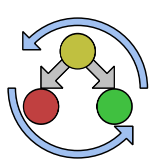
Tree Rotations

- While the binary search tree is a useful data structure…
- …they can degenerate from O(log n) to O(n)
- Fortunately, there are a number of techniques can be used to maintain balance

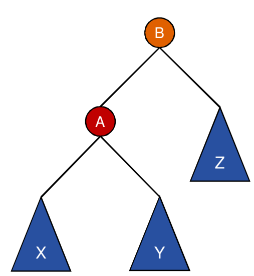


Tree Rotations

- One simple technique (that we will use later) is *rotation*
- When nodes are rotated, they are rearranged in such a way that preserves the BST logic
- We can (and will) rearrange them to preserve O(log n)

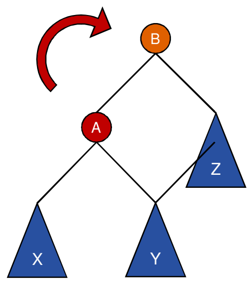


Observe this tree…

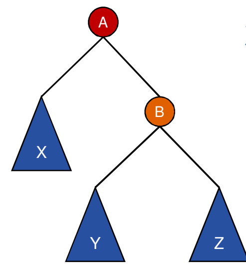


X < A < Y < B < Z

Rotation…

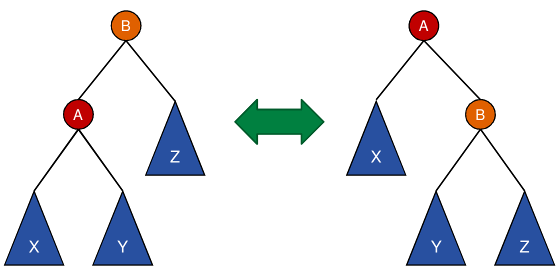


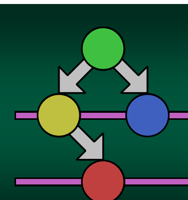
Same tree



X < A < Y < B < Z

Rotations



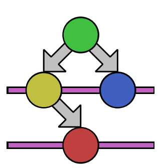


AVL Trees

Bringing balance… aggressively

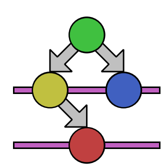
AVL Trees

- *AVL Tree* is a height-balanced binary search tree invented by Adelson-Velskii and Landis in 1962
- It was the first tree balancing algorithm – 8 years before 2-3 Trees were invented



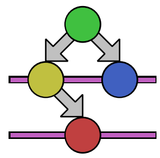
AVL Trees

- The ADT keeps track of the height of each subtree and reorders the data as needed
- AVL Trees aggressively balance the nodes – which ensures the O(log n) search



AVL Trees

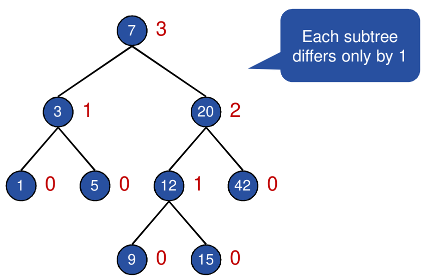
- So, searching is always optimized
- However, adding nodes requires considerable work and, ultimately, hurts efficiency



AVL Trees

- Each subtree has a "height" property
  - it is the maximum between the height of the left and right subtree + 1
  - leafs have a height of zero
- If the right and left branches only differ by **1**, the AVL Tree is sufficiently balanced
- If not, they are balanced by rotating

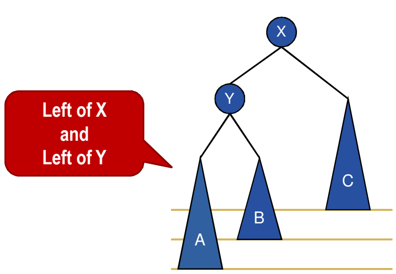
Subtree Heights



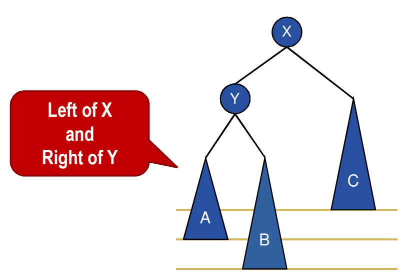
Inserting Nodes

- Unless values are inserted in a very specific order, the tree will, naturally, become unbalanced
- Imbalance falls into two distinct categories
  1. Left-Left (or Right-Right) imbalance
  2. Left-Right (or Right-Left) imbalance

Left-Left Imbalance



Left-Right Imbalance

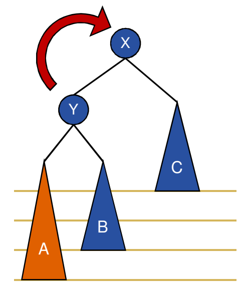


Insert and Rotate

- When a node is inserted… only nodes on the path from insertion point to the root have possibly changed in height
- So after the Insert…
  - start balancing starting at the lowest node
  - recurse back up to the root rotating *as needed*

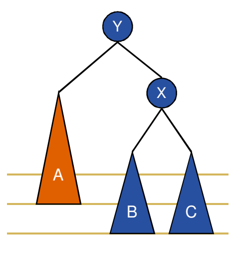
Left-Left Imbalance

- Children of X differ by more than 1
- A's height is 1 larger than B and C
- Rotate right…
  - make Y the new root
  - X its right child of Y
  - B and C subtrees of X

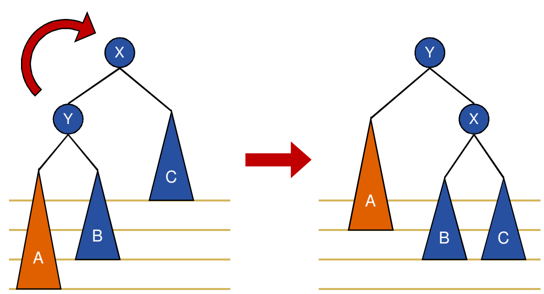


Left-Left Imbalance

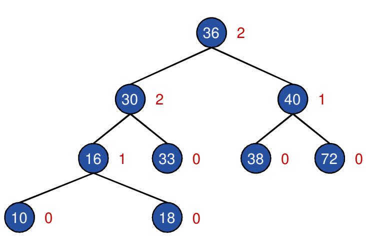
- After the rotation, B and C now have the same height
- A's is one less than B and C
- But the difference is now 1 – the tree is balanced



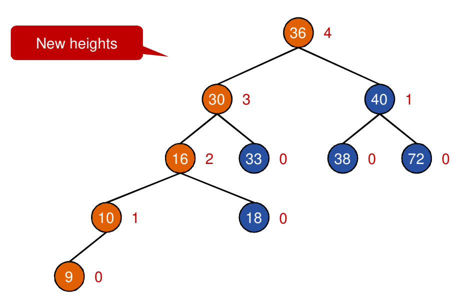
Left-Left Rebalance



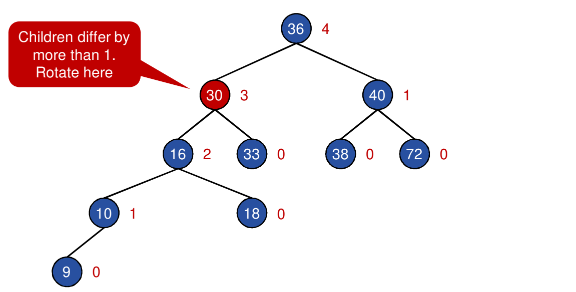
Example: Add 9



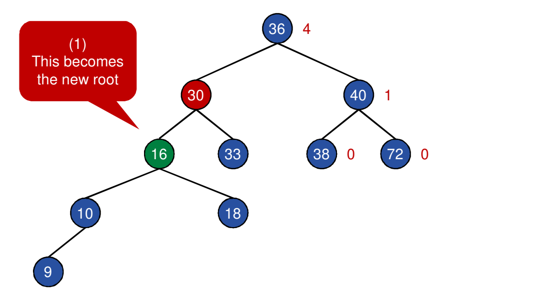
Example: Add 9



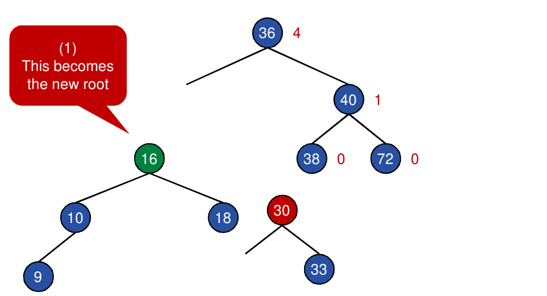
Example: Left-Left Unbalance



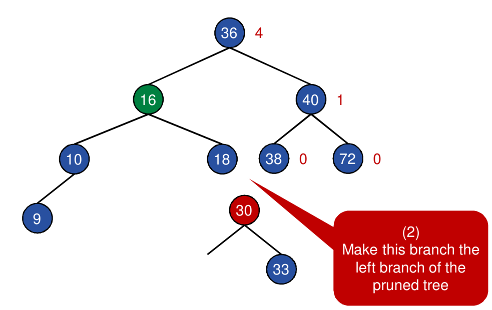
Example: Left-Left Unbalance



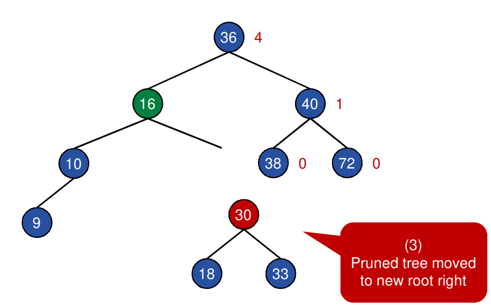
Example: Left-Left Unbalance



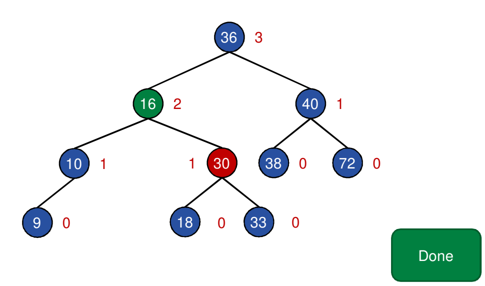
Example: Left-Left Unbalance



Example: Left-Left Unbalance

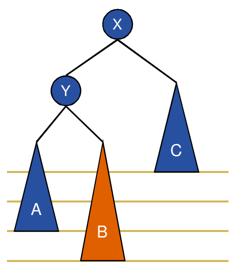


Example: Left-Left Unbalance



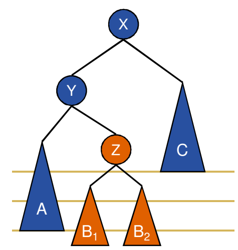
Left-Right Imbalance

- Can't use the Left-Left balance trick - because now it's the *middle subtree*, i.e. B, that's too deep.
- Instead consider what's inside B...



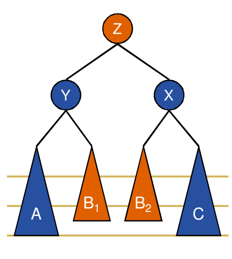
Left-Right Imbalance

- B will have two subtrees containing at least one item (just added
- We do not know which is too deep - set them both to **0.5** levels below subtree A

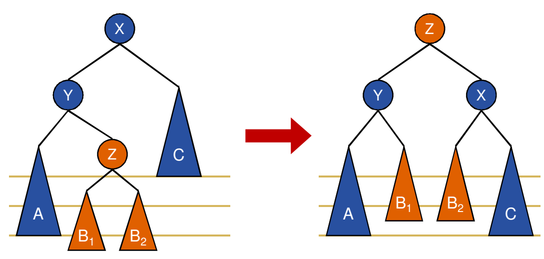


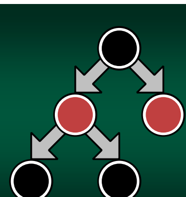
Left-Right Imbalance

- Neither X nor Y worked as root node so make Z the root
- Rearrange the subtrees in the correct order
- No matter how deep B1 or B_2 (+/- 0.5 levels) we get a legal AVL tree again



Left-Right Rebalance



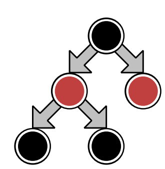


Red-Black Trees

Bringing Balance with Ease

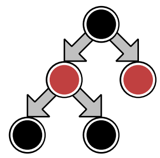
Red-Black Trees

- **Red**-**Black** trees were invented by Rudolf Bayer in 1972
- While 2-3 Trees are amazing, the nodes are a tad complex
- Can they be implemented by only using 2-Nodes?
- The answer is: ***yes!***

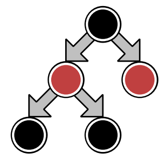


Red-Black Trees

- The *Red-Black Tree* is ADT that implements a 2-3 tree using strictly 2-nodes
- However, this does add some complexity to our logic… but we have the same results: *balance*

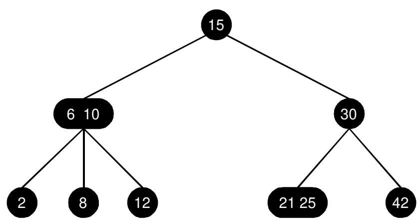


Red-Black Trees

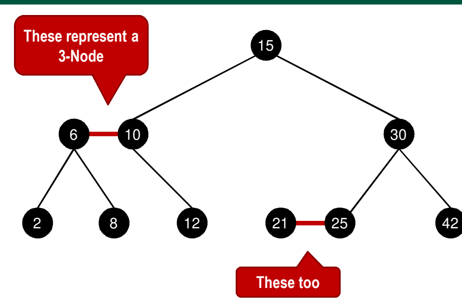


- So, let's look a 2-3 tree and make some modifications
- First, we will convert all of our 3-nodes into a chain of two 2-Nodes
- So we know that they belong together, let's mark the branch as **red**

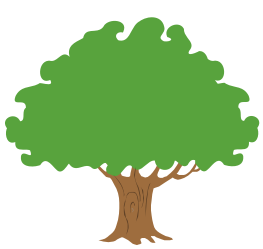
Basic 2-3 Tree



Represented with only 2-Nodes

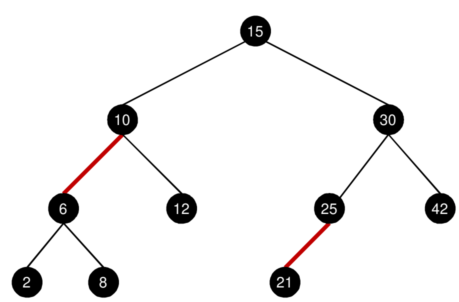


Red-Black Trees

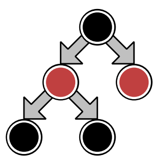


- Of course, we don't typically represent trees using horizontal links
- So, let's rearrange the nodes into a typical tree structure

Same tree – normal layout

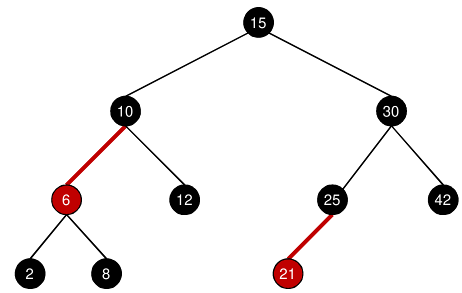


Coloring the node

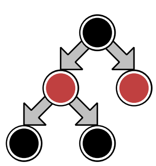


- Naturally, we can't color branches (which are just references/links in Java)
- … or any major language
- We can color the nodes, that are children of the red-branch, as **red**

Coloring the Nodes Red

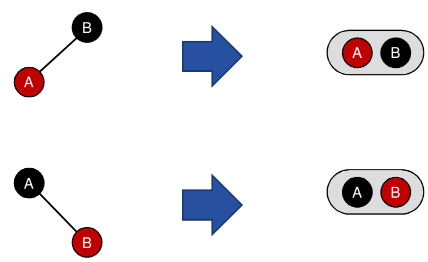


2-3 Trees in Red and Black

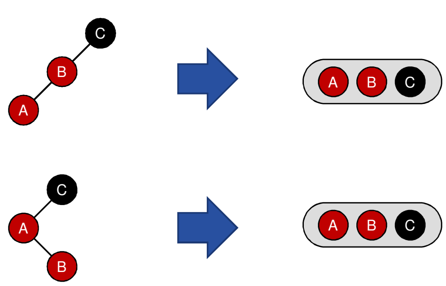


- So, a Red-Black tree is basically a 2-3 Tree stored using only 2-nodes
- So, think of a red node as part of the parent

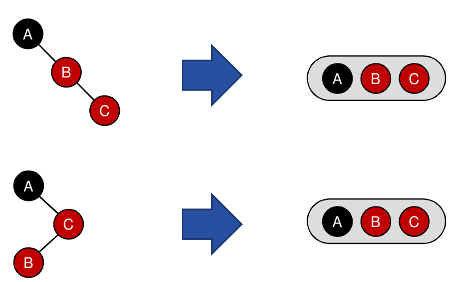
3-Nodes in a Red-Black Tree



4-Nodes in a Red-Black Tree

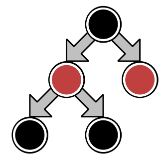


4-Nodes in a Red-Black Tree



4-Nodes in a Red-Black Tree

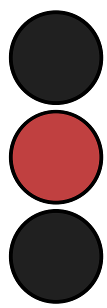
2-3 Trees in Red and Black



- We can get the same advantages as 2-3 trees, but the logic is going to get more complex
- We will handle splits using rotations (like in AVL trees)
- Like 2-3 trees, we will only add at the root

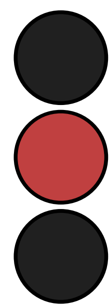
Red-Black Tree Definition

- In a *Red-Black Tree*, every node is marked as either **Red** or **Black**
- Colors were arbitrarily chosen
  - there is no metaphor
  - they looked best on laser printers at the time



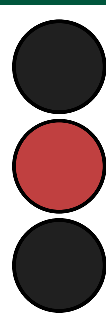
Red-Black Tree Definition

- If a node is **Red** then both children are **Black**
- That makes sense, or it would be representing a 4-Node (or something even larger)



Red-Black Tree Definition

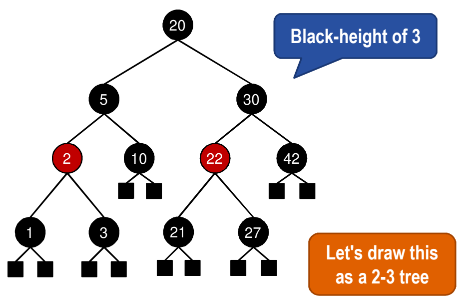
- The root considered **Black**
- Null pointers are considered **Black** nodes
  - even though they are not really nodes…
  - the algorithm uses this logic



The Black-Height

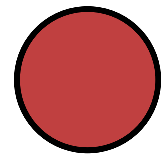
- *Black-height* of a node is the number of **Black** nodes on any path to a null
- We don't count red nodes since they are represent part of a 3-Node
- Typically, the root isn't counted
- Every path from any node to a null contains the same number of **Black** nodes

Black-heights



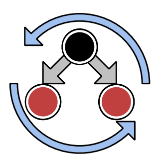
Adding Nodes

- When an node is added, in a 2-3 Tree, it is merged into the leaf node
- So, in Red-Black trees:
  - new node is added as a leaf
  - …but is part of the parent
  - so, all added nodes are **red**

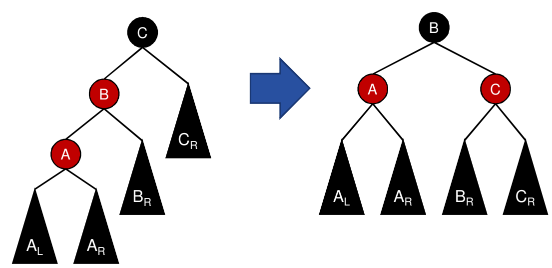


Balancing the tree

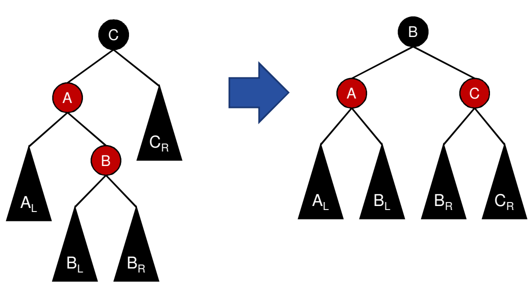
- Additional rotations are done to avoid 2 red nodes in a row
- The rotations change the tree to have a **red**-**black**-**red** pattern → a 4-node
- We will also recolor the nodes afterwards



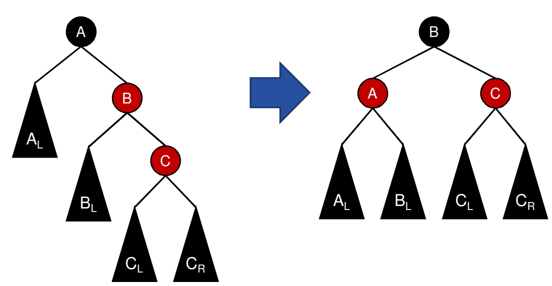
Red-Black Rotation – Case 1



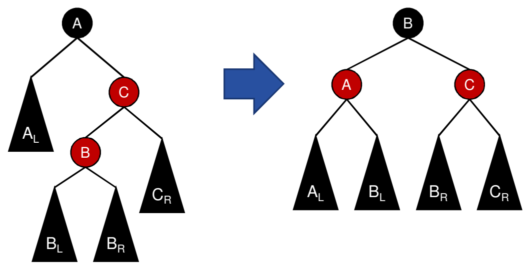
Red-Black Rotation – Case 2



Red-Black Rotation – Case 3



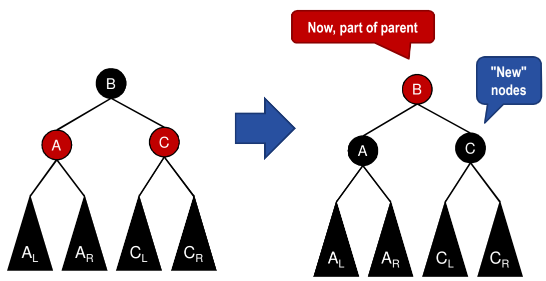
Red-Black Rotation – Case 4



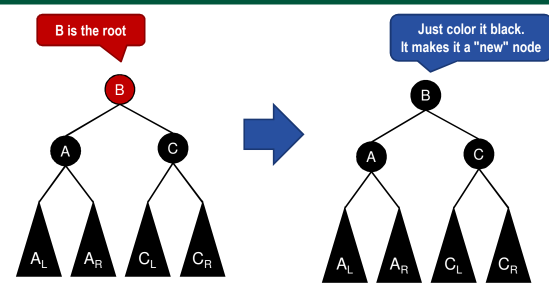
Splitting in the Red-Black Tree

- Splitting in a Red-Black Tree is **amazingly** simple
- It works when a black node has two red children
- Remember…
  - a red node is part of its parent
  - so, we simply need to recolor the nodes!

Splitting in the Red-Black Tree

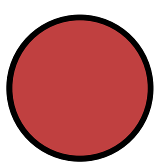


Red Roots are Colored Black

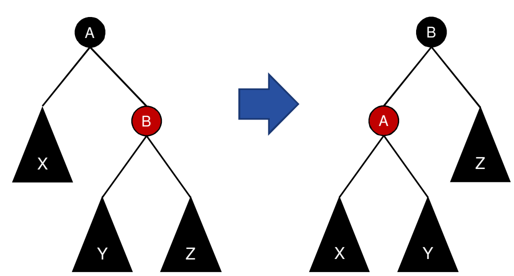


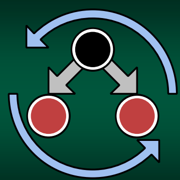
Reds are Usually Left Children

- Most implementations of Red-Black trees maintain the red nodes as left-children
- This simplifies some of the logic later
- So, when a red right-child is added, the nodes are rotated



Rotate Red to Left-Child



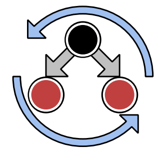


Red-Black & 2-3 Tree Comparison

They Are The Same

Red-Black & 2-3 Tree Comparison

- The follow section builds a tree side-by-side using a Red-Black and 2-3 Tree
- Note that both trees are always conceptually identical
- … though stored differently



Red-Black vs 2-3 Comparison

74	36	10	5	20	42

Red-Black Tree 2-3 Tree

Add 74

74	36	10	5	20	42

Red-Black Tree 2-3 Tree

Add 36

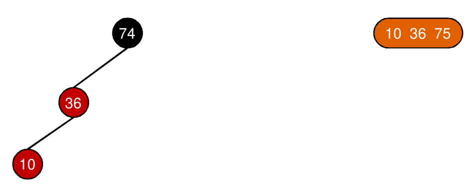
74	36	10	5	20	42

Red-Black Tree 2-3 Tree

Add 10: Rotation Needed

74	36	10	5	20	42

Red-Black Tree 2-3 Tree

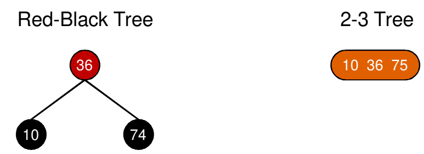


Add 10: Rotation Complete

74	36	10	5	20	42

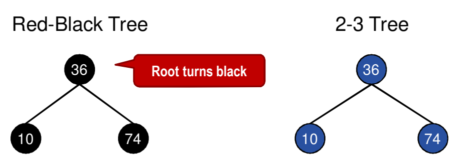
Add 10: Split Red-Black Node

74	36	10	5	20	42



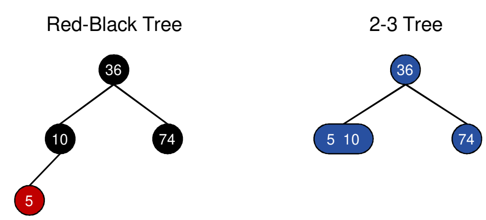
Add 10: New Root Created

74	36	10	5	20	42



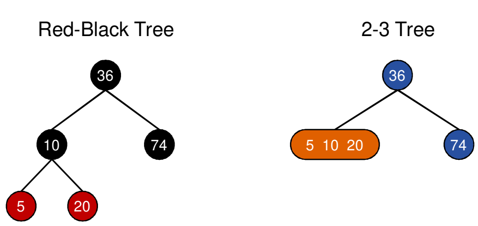
Add 5

74	36	10	5	20	42



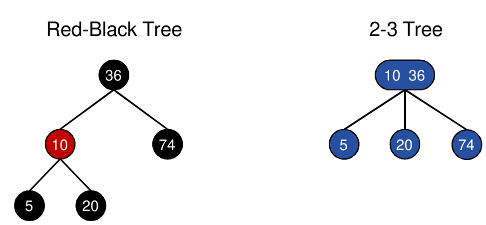
Add 20 – Split Needed

74	36	10	5	20	42



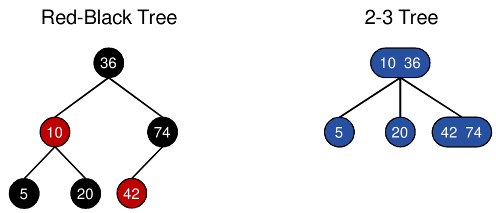
Add 20 – Split Complete

74	36	10	5	20	42



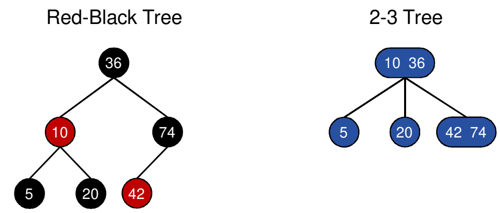
Add 42

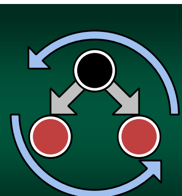
74	36	10	5	20	42



Complete

74	36	10	5	20	42



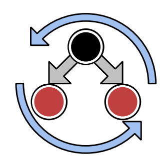


Real-World Red-Black Rotations

Just Skip A Few Steps

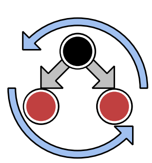
Red-Black & 2-3 Tree Comparison

- In the previous section, all the rotations put the tree into a **red-black-red** format
- …this this was "split" by making it **black-red-black**

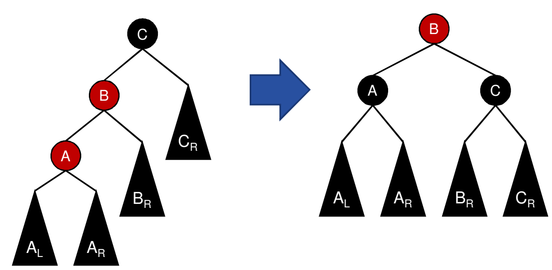


Red-Black & 2-3 Tree Comparison

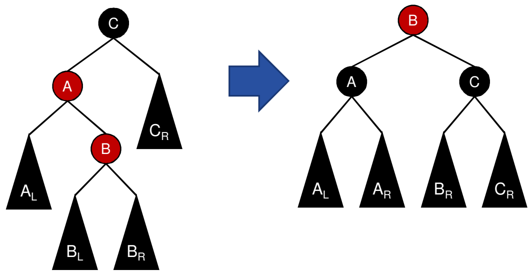
- In reality, the **red-black-red** step can be skipped (since immediately it will be split)
- So, in real-world Red-Black trees, we rotate and split in the same move



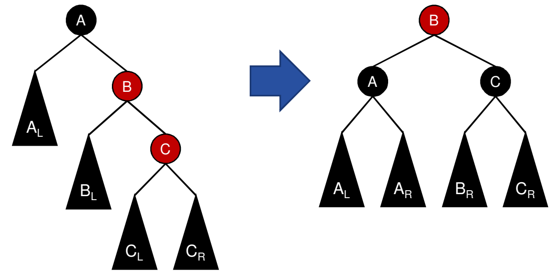
Red-Black Rotation – Case 1



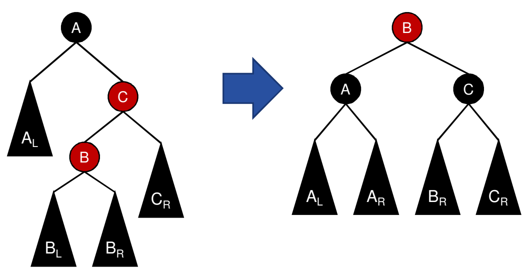
Red-Black Rotation – Case 2



Red-Black Rotation – Case 3



Red-Black Rotation – Case 4



Red-Black Recolor – Case 5