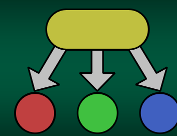
Balanced Trees

Part 11

1



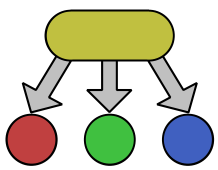
2-3 Trees

Balance using really big nodes

2

2-3 Trees

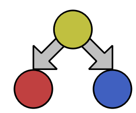
- The *2-3 Tree* is a special type of BST invented by *John Hopcroft* in 1970
- *It automatically maintains balance as it grows!*
- It does this by using a clever variation of the node that can contain multiple values



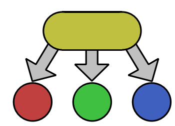
3

2-3 Trees

- *2-Nodes*
  - contains 1 value
  - two children: left and right



- *3-Nodes*
  - contains **2** values
  - three children: left, middle, and right



4

Searching a 2-3 tree

- Searching a 2-3 Tree is very similar to a Binary Search Tree, but with a minor difference
- Both are easy to code and traversal logic is straight forward

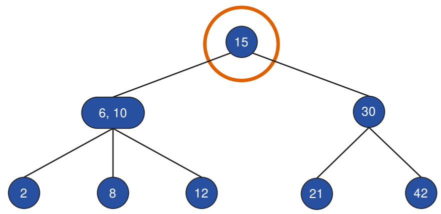
5

Searching a 2-3 tree

- 2-nodes:
  - if less than → go left
  - if greater than → go right
- 3-nodes for values *a*, *b*:
  - if less than *a* → go left
  - if between *a* and *b*, → middle
  - if greater than *b* → go right

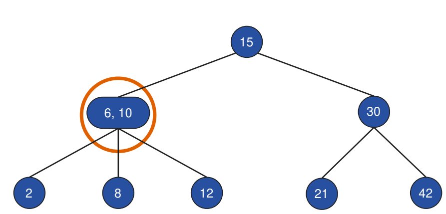
6

Search for 8: Go Left



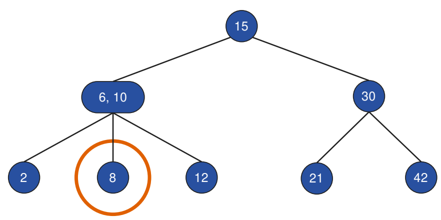
7

Search for 8: Go Middle



8

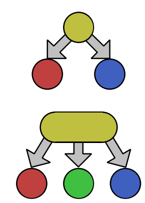
Search for 8: Found



9

Adding to a 2-3 tree

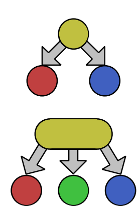
- For BSTs, when a value is added, it will create a new left or right leaf
- 2-3 Trees, however, will merge the value into the leaf (rather than a new node)



10

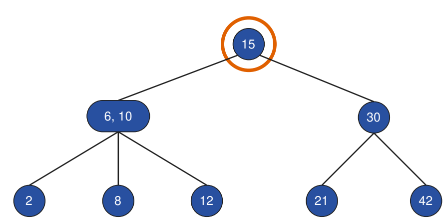
Adding to a 2-3 tree

- 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…



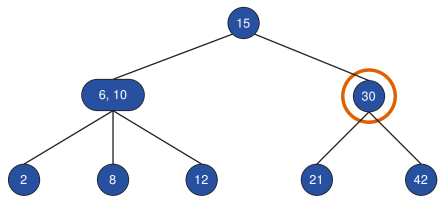
11

Add 25: Go Right



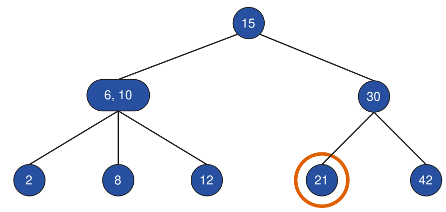
12

Add 25: Go Left



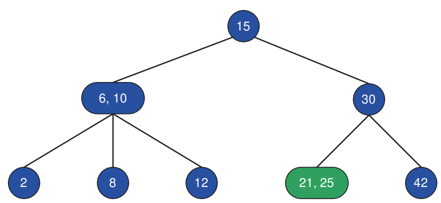
13

Add 25: Can't go further



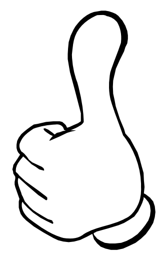
14

Add 25: Convert 2-Node to 3-Node



15

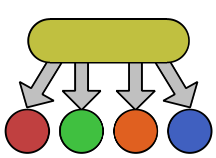
Adding to a 2-3 tree



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

16

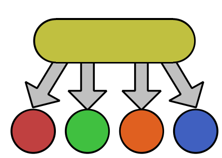
The 4-Node



- 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

17

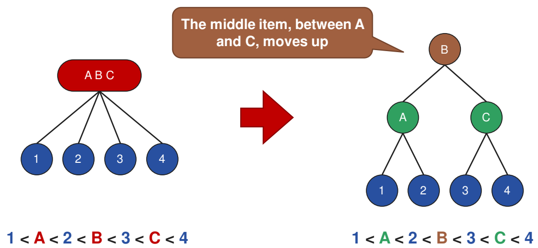
The 4-Node



- 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, it can split equally
- … and balanced!

18

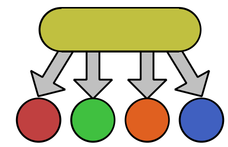
One Way to Split a 4-Node



19

The Types of Splits

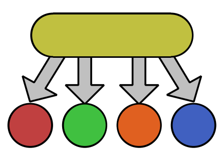
- There are a total of six different splits that can occur in a 2-3 tree
- In each split, the **middle** value **ascends** up to the parent node



20

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)



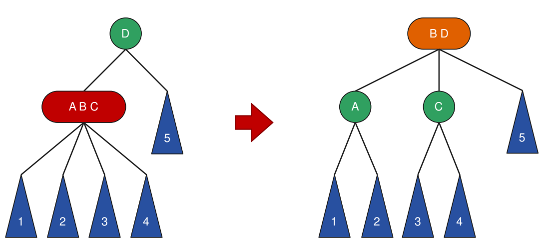
21

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)

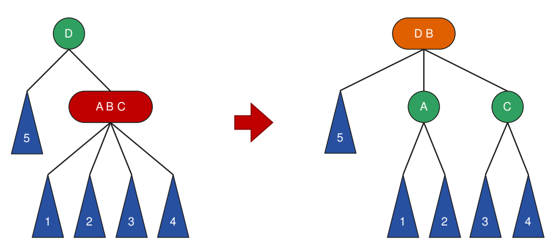
22

2-Node Parent, Left Child (1)



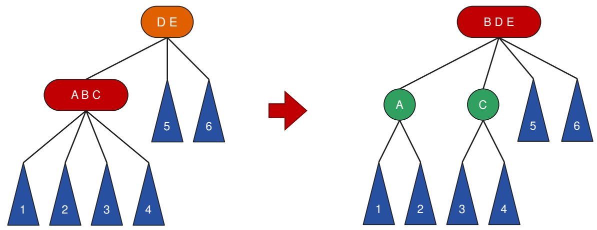
23

2-Node Parent, Right Child (2)



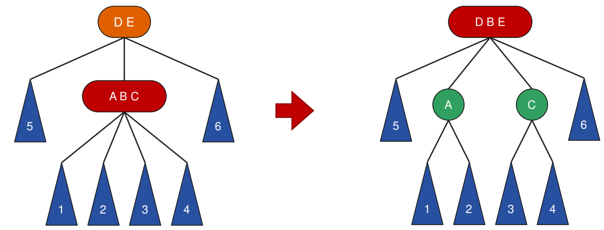
24

3-Node Parent, Left Child (3)



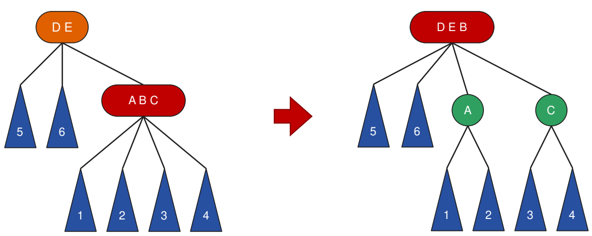
25

3-Node Parent, Middle Child (4)



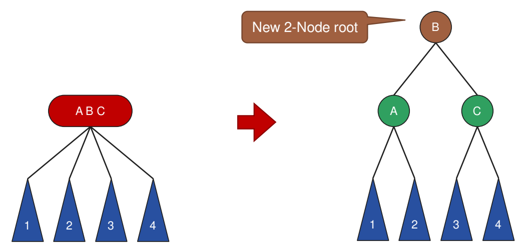
26

3-Node Parent, Right Child (5)



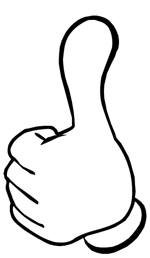
27

Node is the root (6)



28

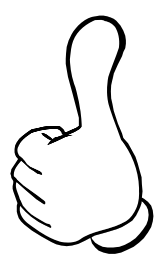
Why Does This Work?



- 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 **<u>only</u>** when the root is split*
- … and it splits balanced on the left and right side!

29

Why Does This Work?



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

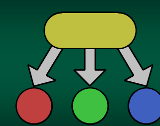
30

Why Does This Work?

- Additionally, 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

Fall 2021 Sacramento State - Cook - CSc 130 31

31



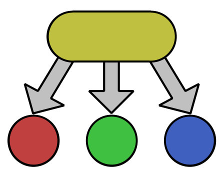
BST & 2-3 Tree Comparison

They Are The Same

32

Basic BST & 2-3 Tree Comparison

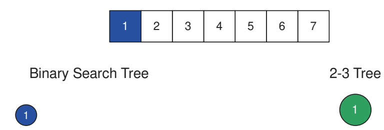
- The follow section builds a tree side-by-side using a basic BST and a 2-3 Tree
- In this case, we will feed a sorted list into the tree
- … which causes a BST to become O(n)



Fall 2021 Sacramento State - Cook - CSc 130 33

33

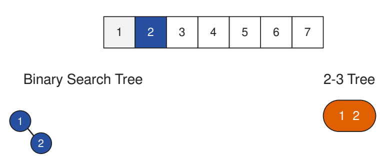
Add 1



Fall 2021 Sacramento State - Cook - CSc 130 34

34

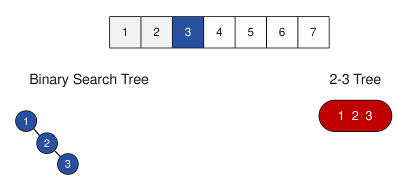
Add 2



Fall 2021 Sacramento State - Cook - CSc 130 35

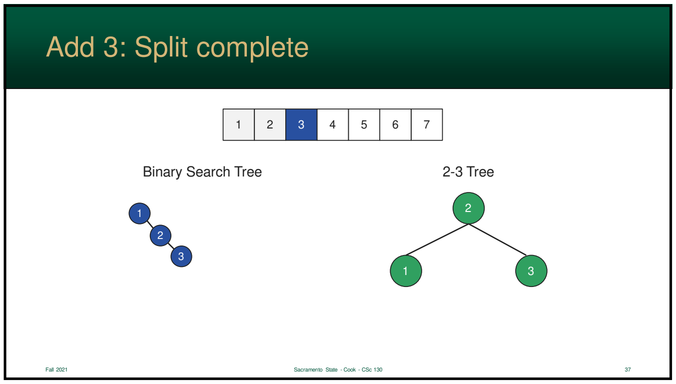
35

Add 3: The 2-3 Tree will split

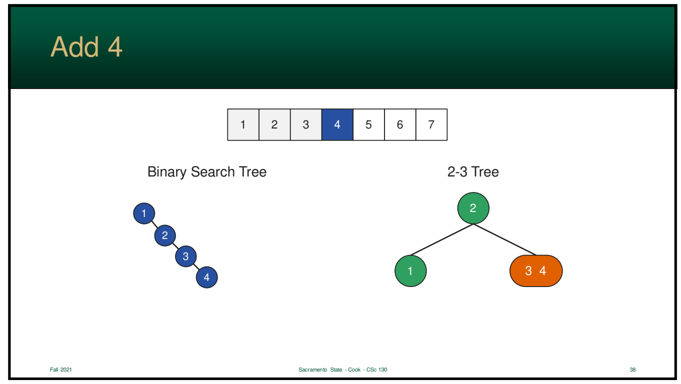


Fall 2021 Sacramento State - Cook - CSc 130 36

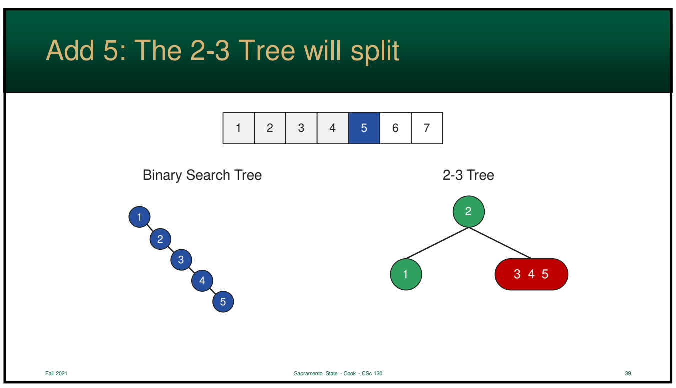
36



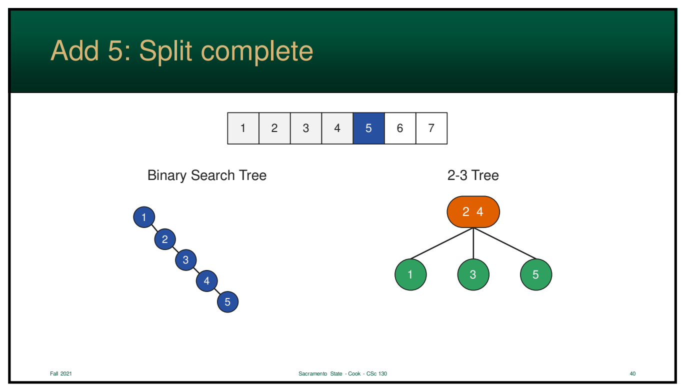
37



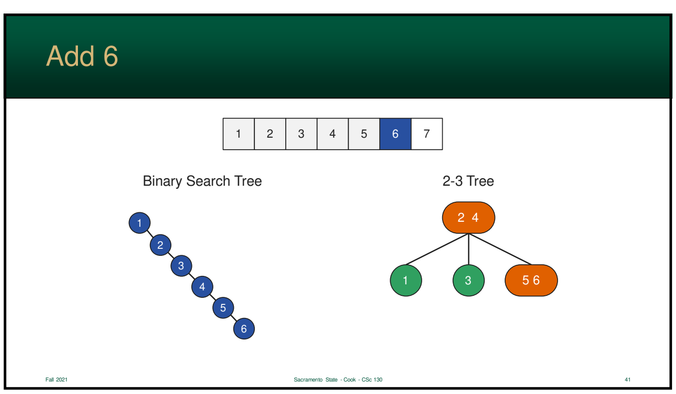
38



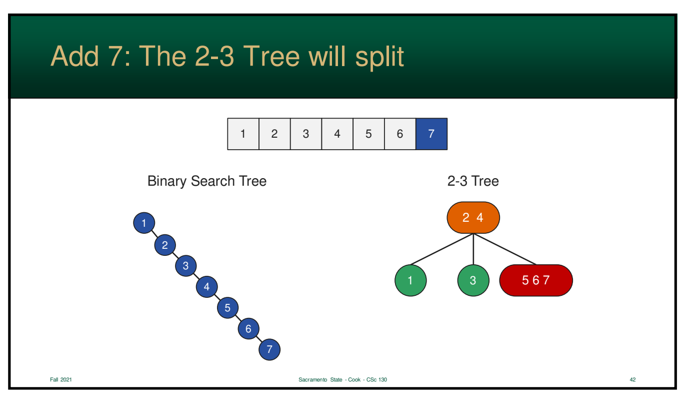
39



40

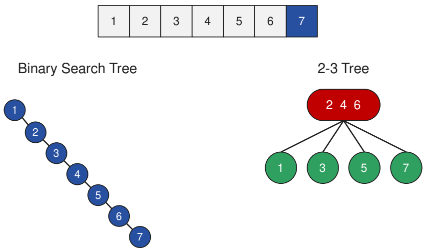


41



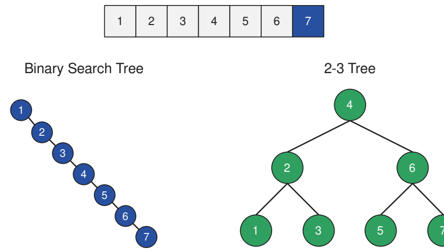
42

Add 7: Ascended value causes split

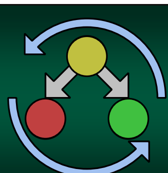


43

Add 7: Split complete



44



Tree Rotations

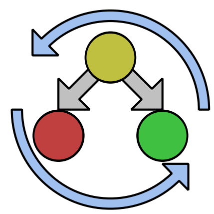
Do-Si-Node

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

45

Tree Rotations

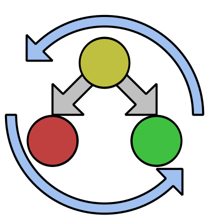
- While the binary search tree is a useful data structure…
- …they can degenerate from O(log n) to **O(n)**
- Fortunately, there are multiple techniques that can be used to maintain balance



46

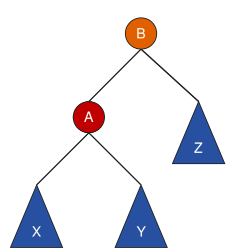
Tree Rotations

- 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)



47

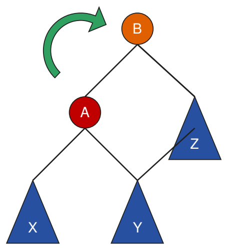
Observe this tree…



X < A < Y < B < Z

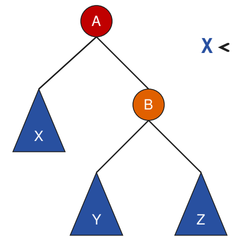
48

Rotation…



49

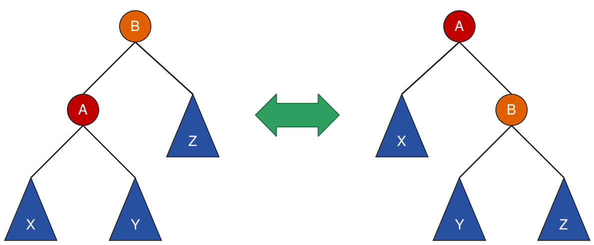
Same tree



X < A < Y < B < Z

50

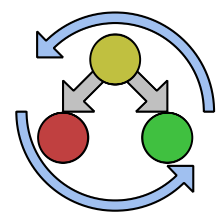
These Trees are Identical



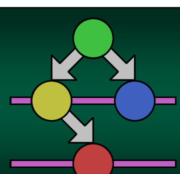
51

These Trees are Identical

- Note that in the last example, the two trees are identical
- However, side with the larger depth, was effectively "shifted" the to other side



52



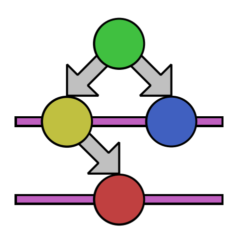
AVL Trees

Bringing balance… aggressively

53

AVL Trees

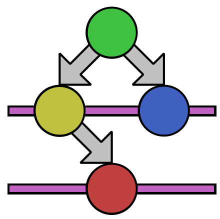
- *AVL Tree* is a height-balanced binary search tree invented by *Adelson-Velskii and Landis* in 1962
- It was the first tree balancing algorithm – 8 years before 2-3 Trees were invented



54

AVL Trees

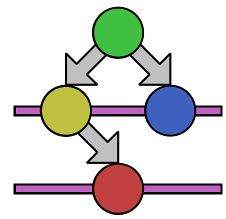
- The ADT keeps track of the height of each subtree and reorders the data as needed
- AVL Trees aggressively balance the nodes – which ensures the O(log n) search



55

AVL Trees

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



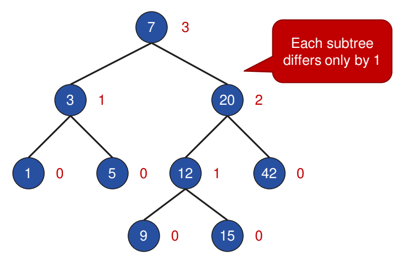
56

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 height of the right and left branches only differ by **1**, the AVL Tree is sufficiently balanced
- If not, they are balanced by rotating

57

Subtree Heights



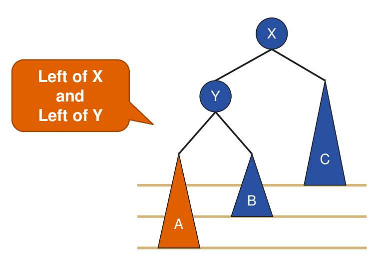
58

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

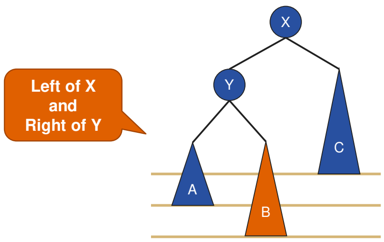
59

Left-Left Imbalance



60

Left-Right Imbalance



61

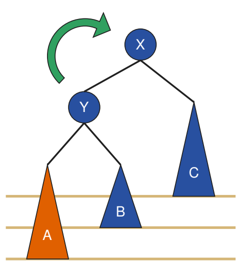
Insert and Rotate

- 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*

62

Left-Left Imbalance

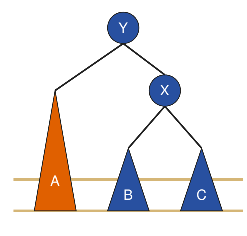
- Children of X differ by more than 1
- A's height is 1 larger than B and C
- Rotate right…
  - Y is the new root
  - X its right child of Y
  - B, C subtrees of X



63

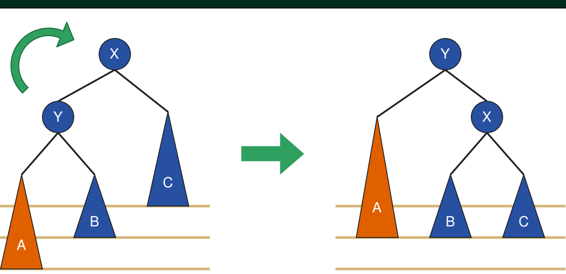
Left-Left Imbalance

- After the rotation, A, B and C have the same height
- Rotation changes the height of the sides by -1 and +1, respectively



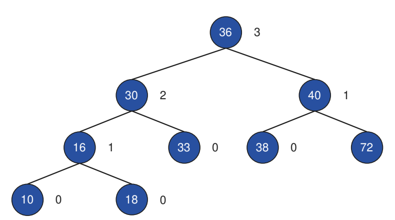
64

Left-Left Rebalance

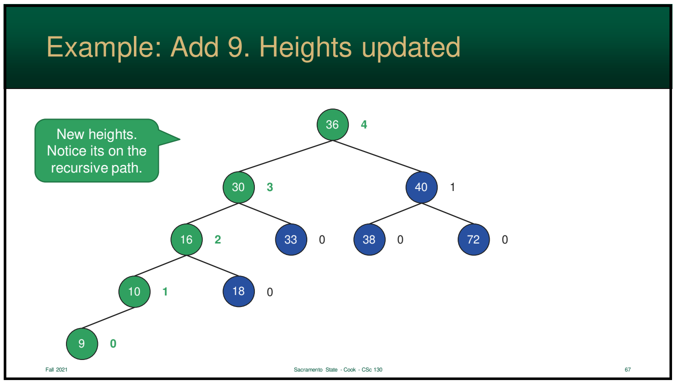


65

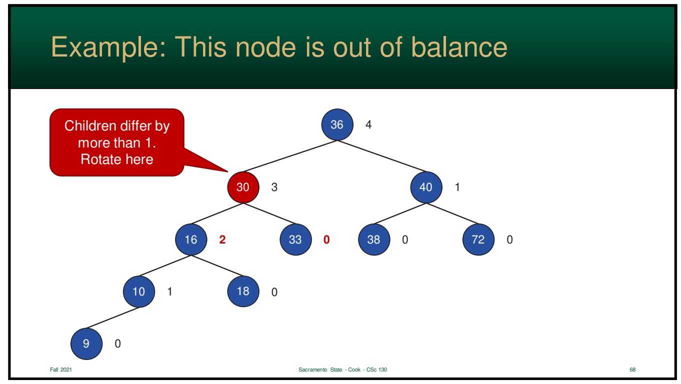
Example: Add 9



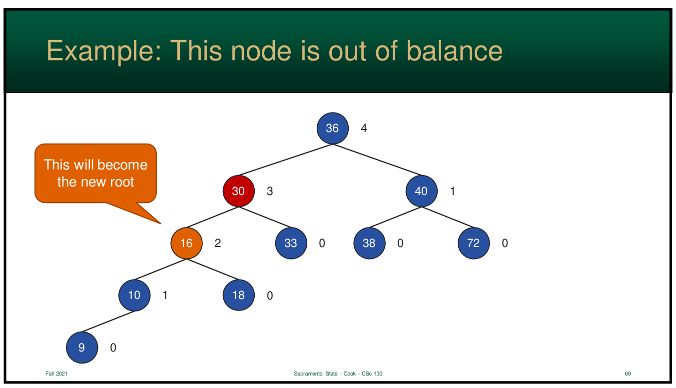
66



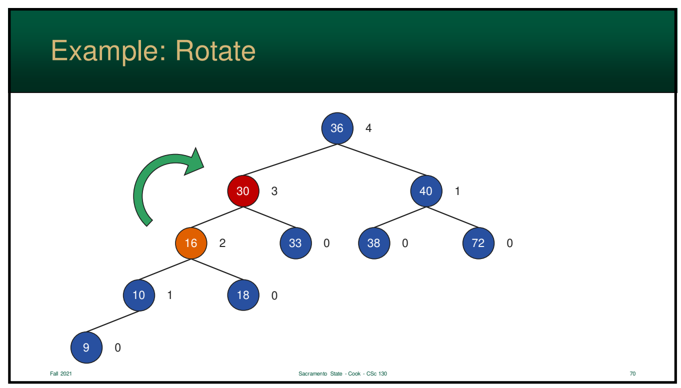
67



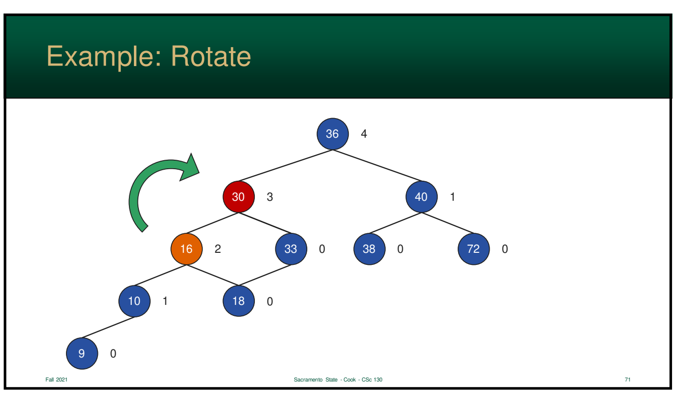
68



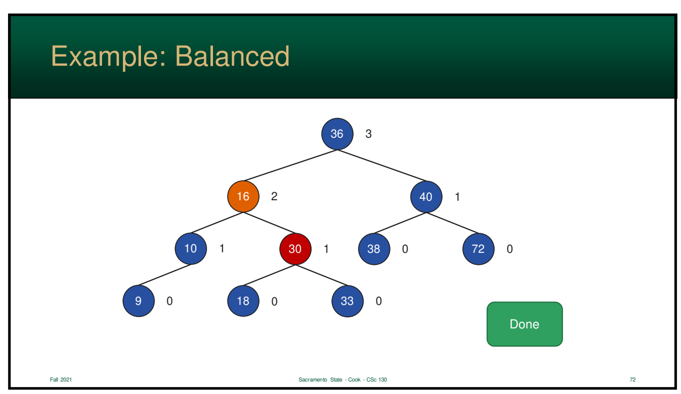
69



70



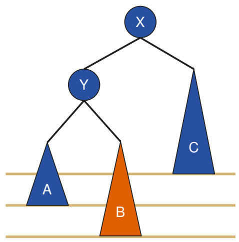
71



72

Left-Right Imbalance

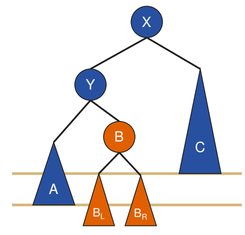
- Can't use the Left-Left balance trick - because now it's the *middle subtree*, i.e. B, that's too deep.
- Instead consider what's inside B...



73

Left-Right Imbalance

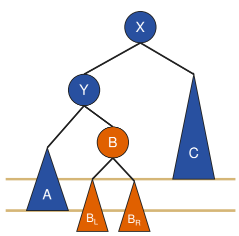
- B will have two subtrees containing at least one item (just added
- We do not know which is too deep - set them both to 0.5 levels below subtree A



74

Left-Right Imbalance

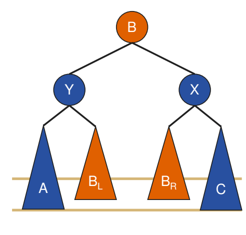
- Neither X nor Y works a new root
- … but look at the value of B
- It is larger than Y, but less than X
- This can be our new root



75

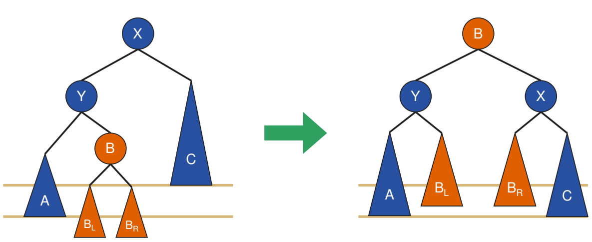
Left-Right Imbalance

- Rearrange the subtrees in the correct order
- No matter how deep B_1 or B_2 (+/- 0.5 levels) we get a legal AVL tree again

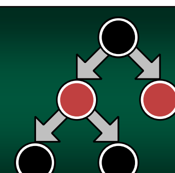


76

Left-Right Rebalance



77



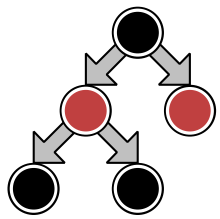
Red-Black Trees

Bringing Balance with Ease

78

Red-Black Trees

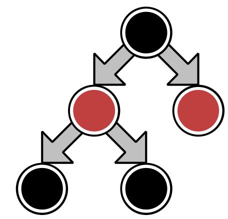
- *Red-Black Trees* are self-balancing BSTs invented by *Rudolf Bayer* in 1972
- 2-3 Trees are amazing, but the nodes are a tad complex
- Can they be implemented by only using 2-Nodes? ***Yes!***



79

Red-Black Trees

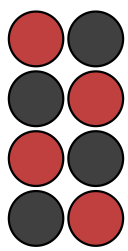
- **Red**-**Black** Tree implements a 2-3 Tree by using strictly 2-nodes
- However, this does add some complexity to our logic… but we have the same results: *balance*



80

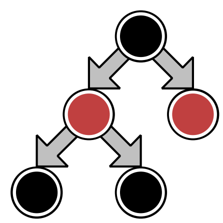
So, Why "Red" and "Black"?

- The colors **Red** and **Black** were arbitrarily chosen
- *Rudolf Bayer* needed a way to mark the nodes differently…
- … these colors looked best on laser printers at the time
- There is **no** metaphor



81

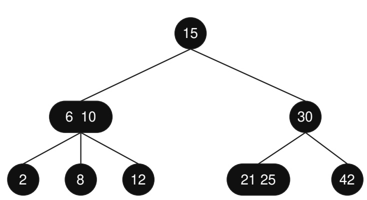
Red-Black Trees & 2-3 Trees



- 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**

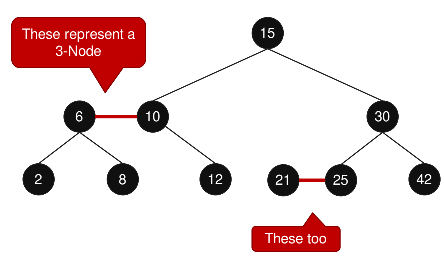
82

Basic 2-3 Tree



83

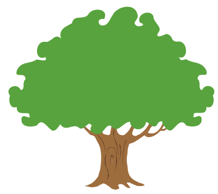
Represented with only 2-Nodes



84

Red-Black Trees

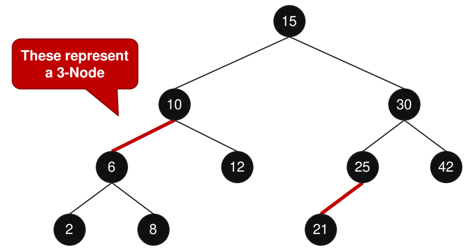
- Of course, we don't typically represent trees using horizontal links
- So, let's rearrange the nodes into a typical tree structure



Fall 2021 Sacramento State - Cook - CSc 130 85

85

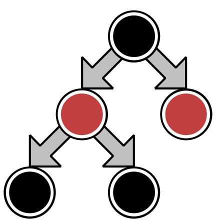
Same tree – Normal Layout



Fall 2021 Sacramento State - Cook - CSc 130 86

86

Coloring the Nodes

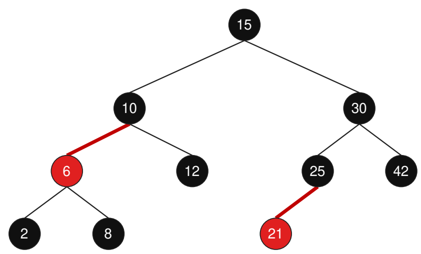


- Naturally, we can't color branches (which are just references/links in Java)
- … or any major language
- We can color the nodes, that are children of the red-branch, as **red**

Fall 2021 Sacramento State - Cook - CSc 130 87

87

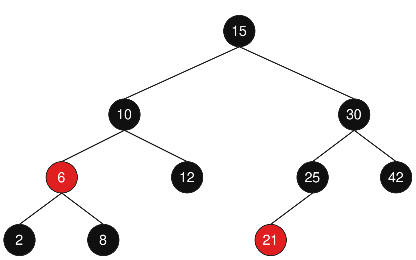
Coloring the Nodes Red



Fall 2021 Sacramento State - Cook - CSc 130 88

88

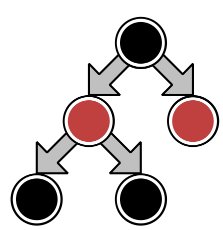
Branches are just branches



Fall 2021 Sacramento State - Cook - CSc 130 89

89

2-3 Trees in Red and Black

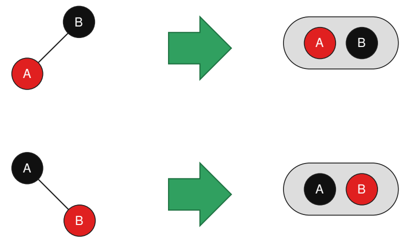


- So, a Red-Black tree is basically a 2-3 Tree stored using only 2-nodes
- *So, think of a red node as part of the parent*

Fall 2021 Sacramento State - Cook - CSc 130 90

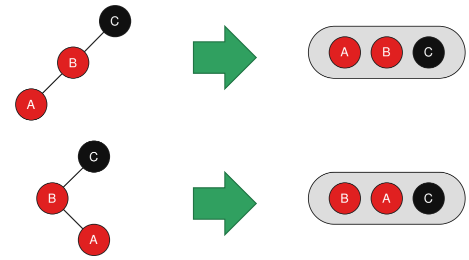
90

3-Nodes in a Red-Black Tree



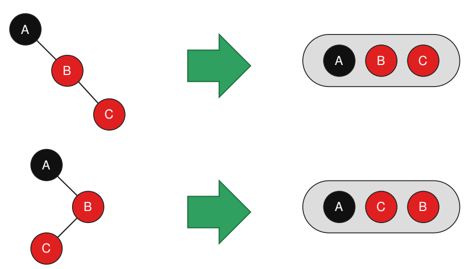
91

4-Nodes in a Red-Black Tree



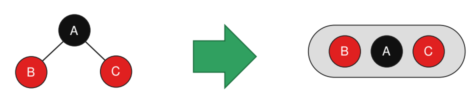
92

4-Nodes in a Red-Black Tree

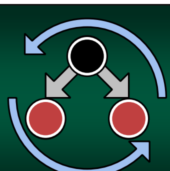


93

4-Nodes in a Red-Black Tree



94

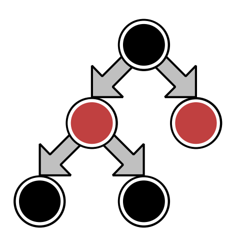


How Red-Black Trees Work

Some of their better known properties

95

2-3 Trees in Red and Black

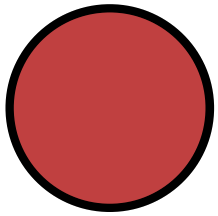


- We can get the same advantages as 2-3 trees, but the logic is going to get more complex
- We will handle splits using rotations (like in AVL trees)
- Like 2-3 trees, we will only add at the root

96

Adding Nodes

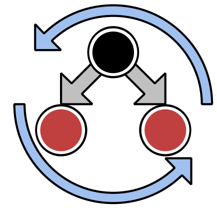
- When an node is added, in a 2-3 Tree, it is merged into the leaf node
- In Red-Black trees:
  - new node is added as a leaf
  - …but it is part of the parent
  - so, all added nodes are **red**



97

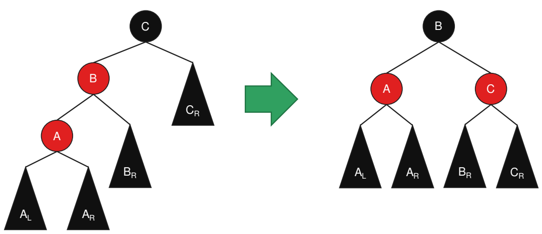
Balancing the tree

- 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



98

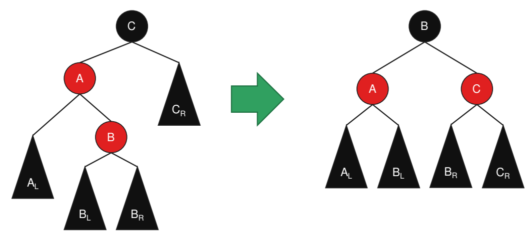
Red-Black Rotation – Case 1



$A_L < A < A_R < B < B_R < C < C_R$ $\quad$ $A_L < A < A_R < B < B_R < C < C_R$

99

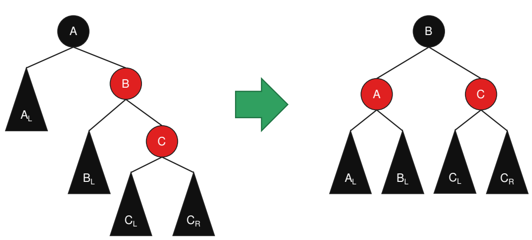
Red-Black Rotation – Case 2



$A_L < A < B_L < B < B_R < C < C_R$ $\quad$ $A_L < A < B_L < B < B_R < C < C_R$

100

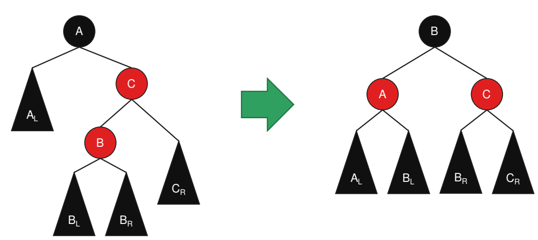
Red-Black Rotation – Case 3



$A_L < A < B_L < B < C_L < C < C_R$ $\quad$ $A_L < A < B_L < B < C_L < C < C_R$

101

Red-Black Rotation – Case 4



$A_L < A < B_L < B < B_R < C < C_R$ $\quad$ $A_L < A < B_L < B < B_R < C < C_R$

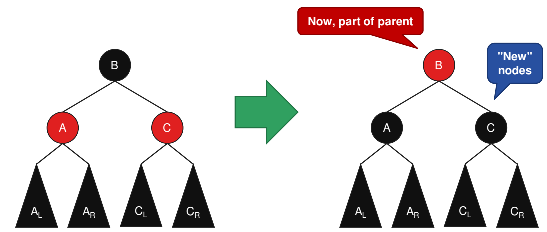
102

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!

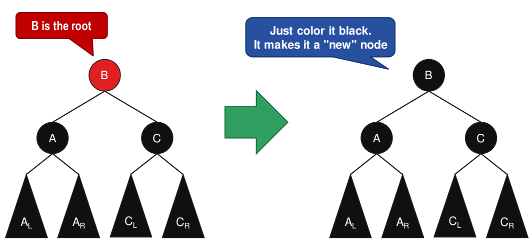
103

Splitting in the Red-Black Tree

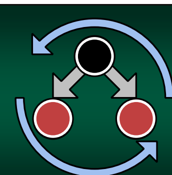


104

Splitting in the Root – Color it Black



105



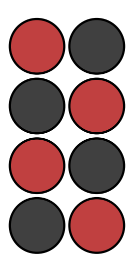
Red-Black Tree Attributes

Some of their better known properties

106

Red-Black Tree Attributes

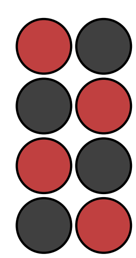
- In a stable tree (not needing rotations), if a node is **Red** then both children are **Black**
- That makes sense, or it would represent 4-Node (or something even larger)



107

Red-Black Tree Attributes

- The root is always considered **Black**
- Null pointers..
  - are considered **Black** nodes - *even though they are not really nodes*
  - typically drawn as rectangles



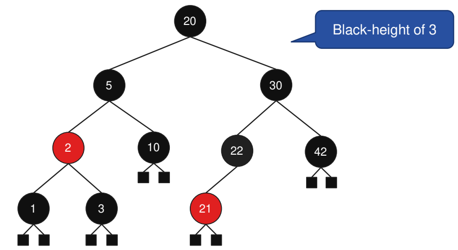
108

The Black-Height

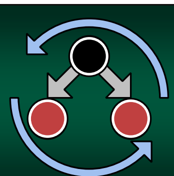
- *Black-height* of a node is the number of **Black** nodes on any path to a null
- We don't count red nodes since they represent part of a 3-Node
- Typically, the root isn't counted
- Every path from any node to a null contains the same number of **Black** nodes

109

Black-heights



110



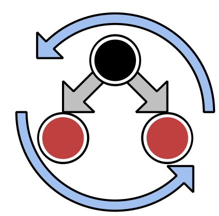
Red-Black & 2-3 Tree Comparison

They Are The Same

111

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



112

Red-Black vs 2-3 Comparison

74	36	10	5	20	42	90

Red-Black Tree 2-3 Tree

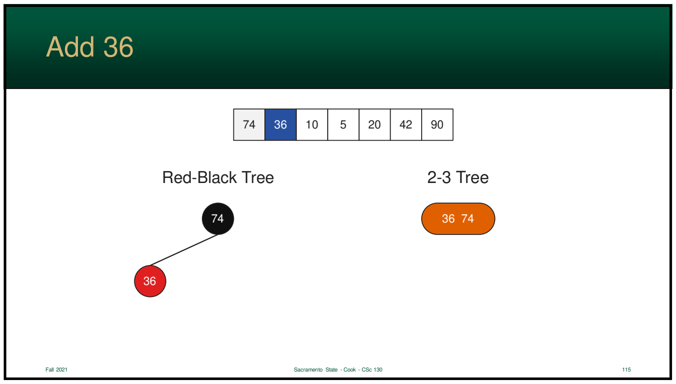
113

Add 74

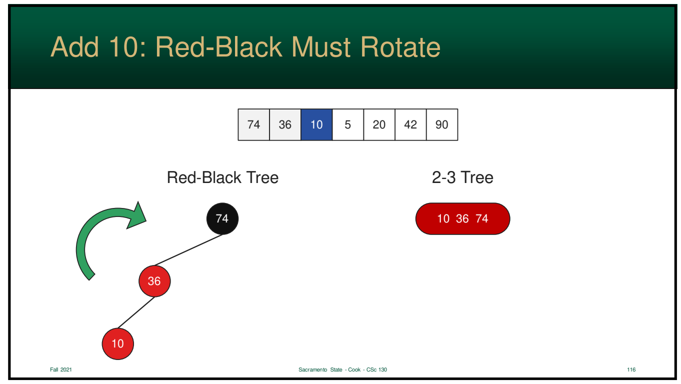
74	36	10	5	20	42	90

Red-Black Tree 2-3 Tree

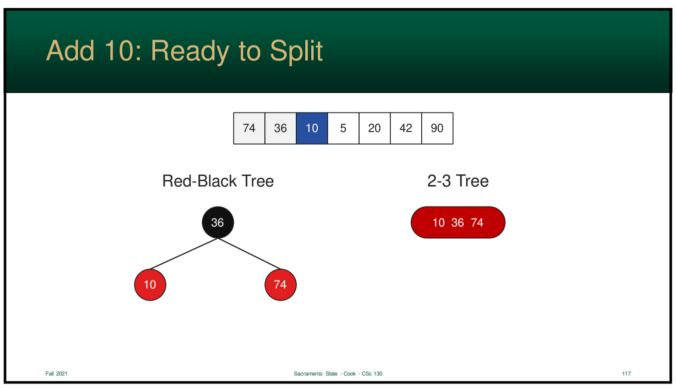
114



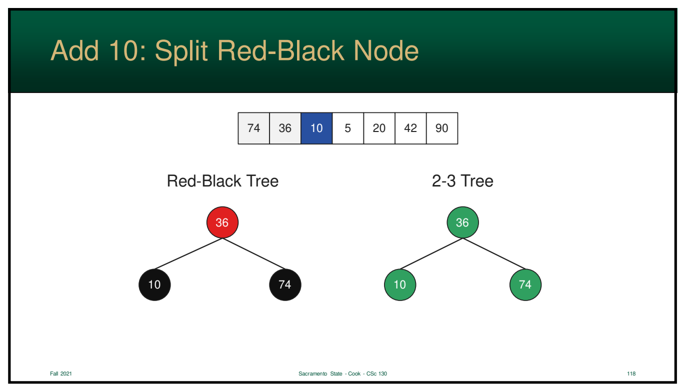
115



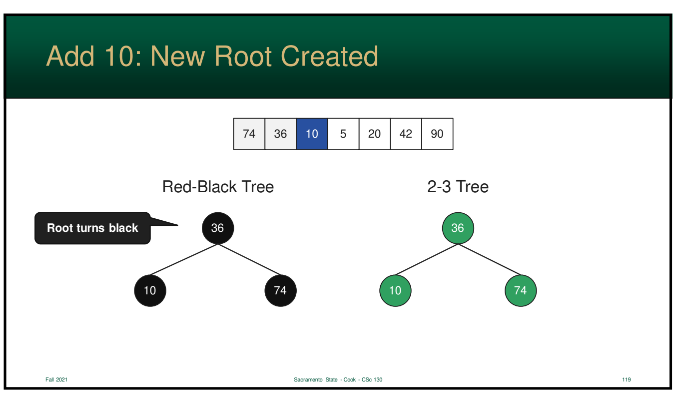
116



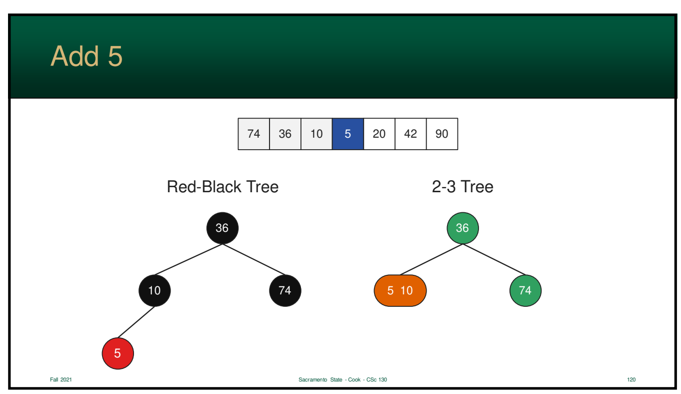
117



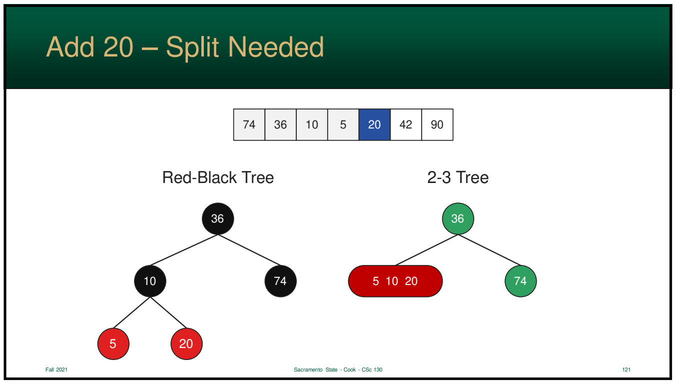
118



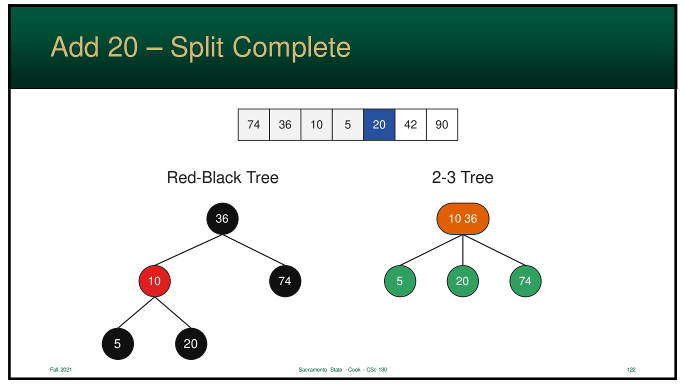
119



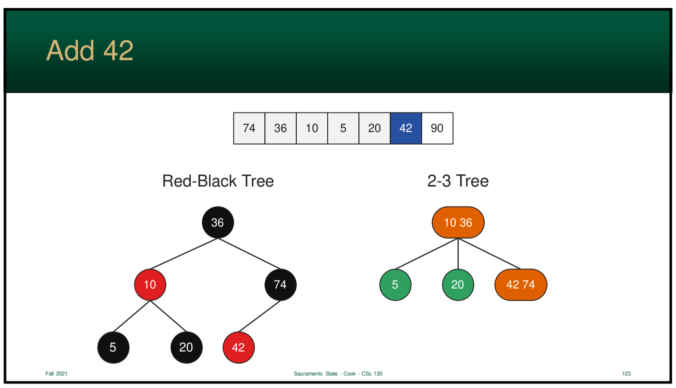
120



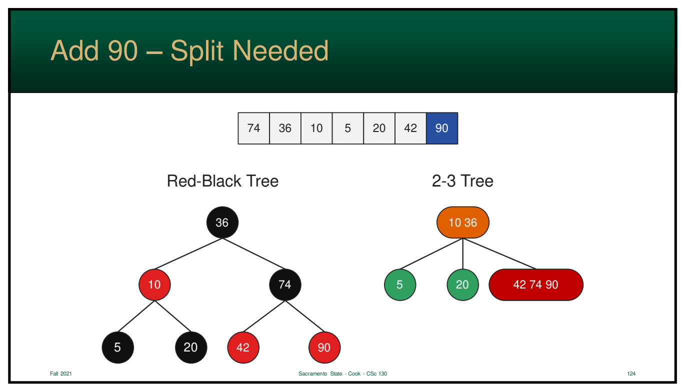
121



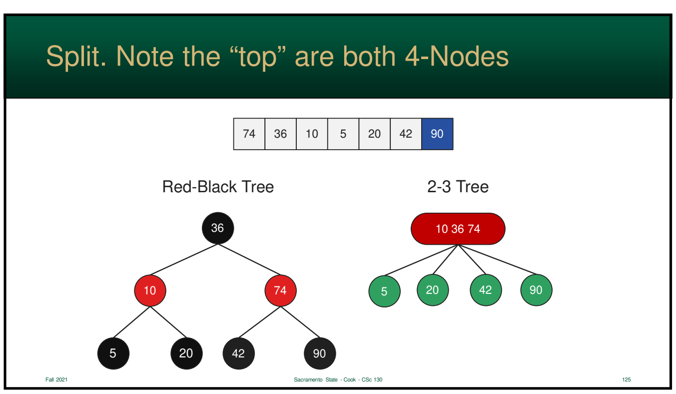
122



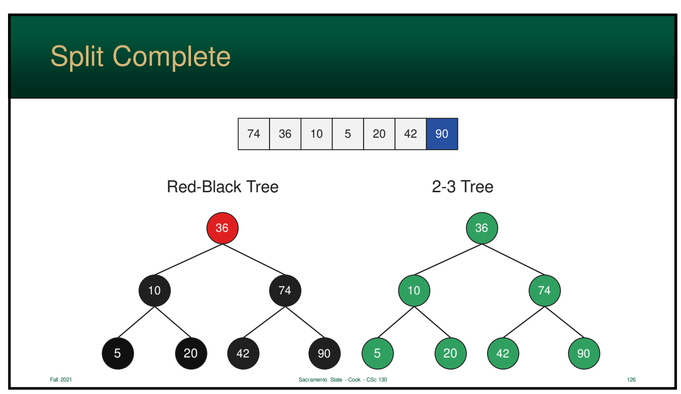
123



124



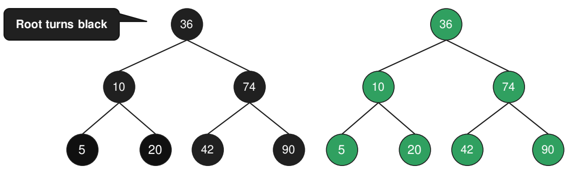
125



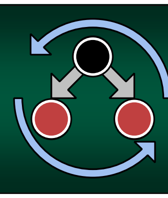
126

Complete

74	36	10	5	20	42	90



127



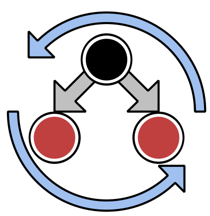
Real-World Red-Black Rotations

Just Skip A Few Steps

128

Red-Black & 2-3 Tree Comparison

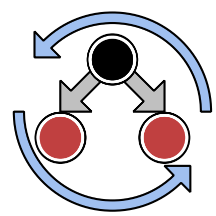
- In the previous section, all the rotations put the tree into a **red**-**black**-**red** format
- …this this was "split" by making it **black**-**red**-**black**



129

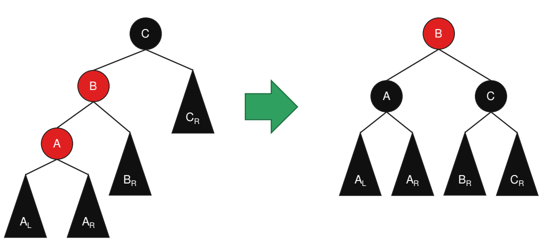
Red-Black & 2-3 Tree Comparison

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



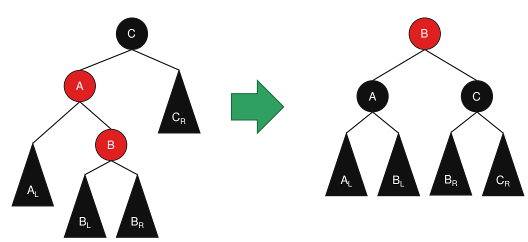
130

Red-Black Rotation – Case 1



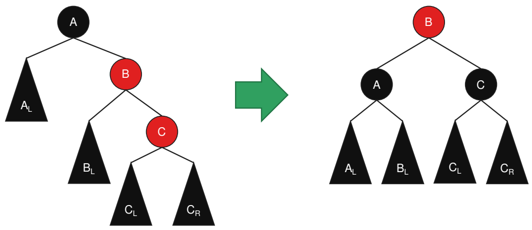
131

Red-Black Rotation – Case 2



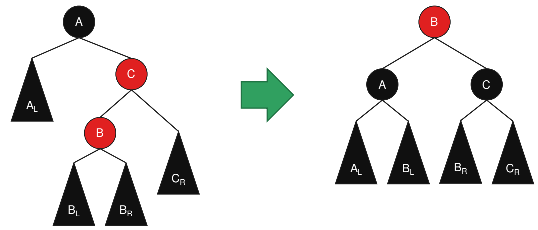
132

Red-Black Rotation – Case 3



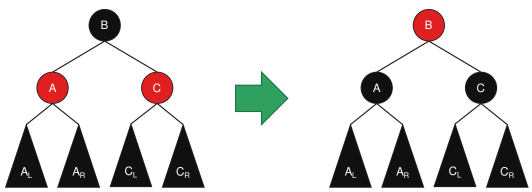
133

Red-Black Rotation – Case 4



134

Red-Black Rotation – Case 5



135

Reds are Usually Left Children

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

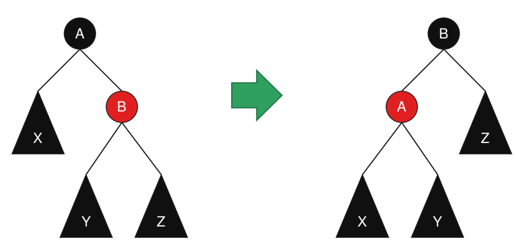
136

Reds are Usually Left Children

- This simplifies the cases – only left-left (case 1) will occur
- Most textbooks and websites will only show you left **red** children

137

Rotate Red to Left-Child



138