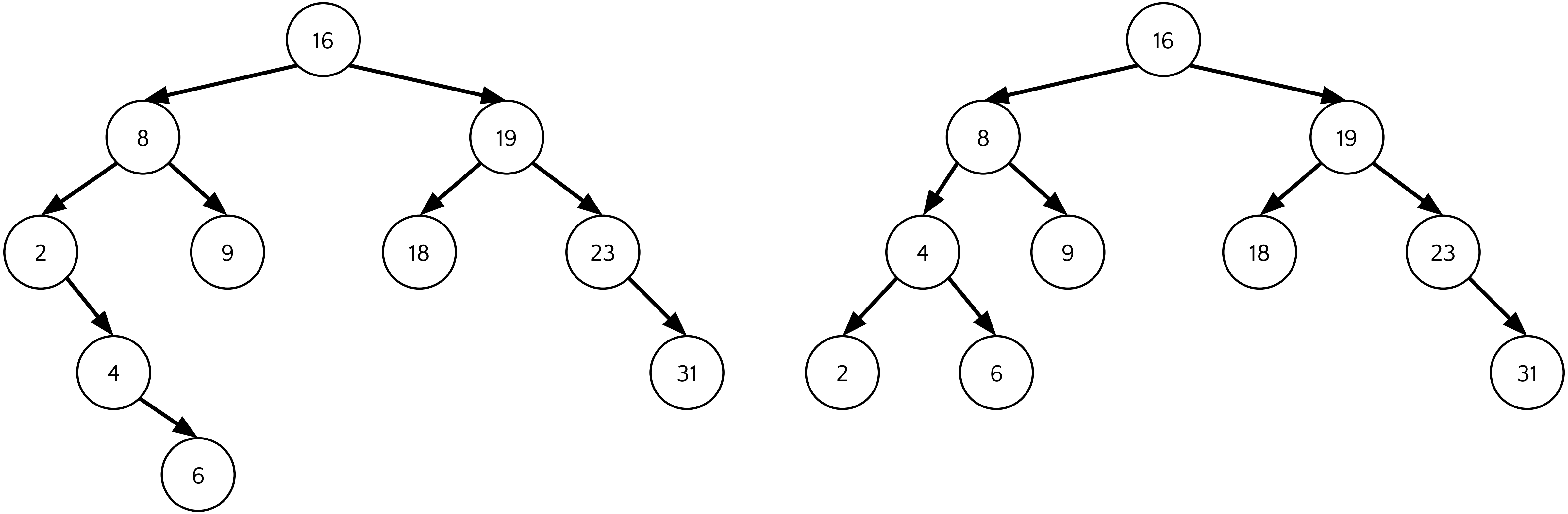
Lecture 11 : Red Black Trees

CSE 373: Data Structures and Algorithms

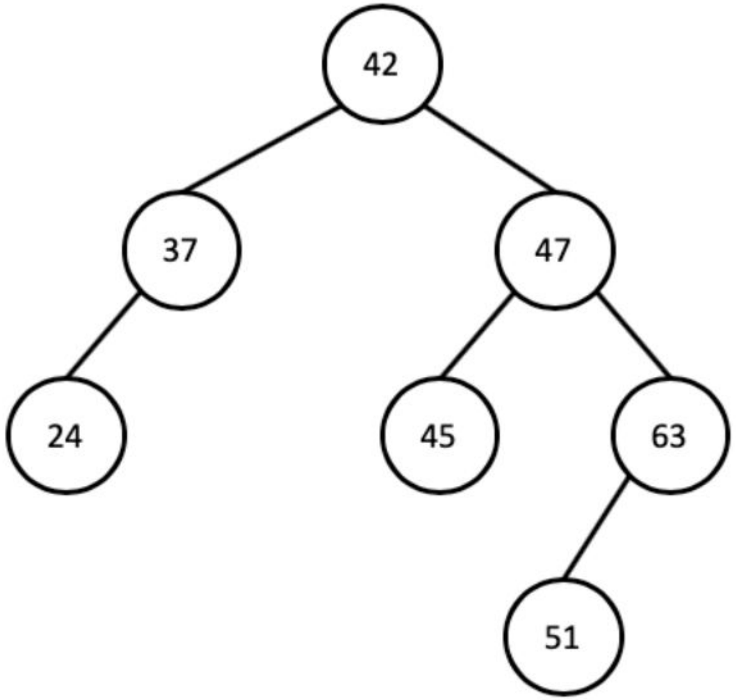
Warm Up

https://app.sli.do/event/8v1Ye3qNk4zJuYXoswAfHq
3586991

What is the final structure of the following AVL tree after inserting 6?



Warm Up



Imagine the value 55 is inserted into the AVL tree shown in Figure 1.

What node becomes 55's parent after insertion and before any rotations?

Which node(s) become imbalanced due to the insertion of 55? This should include all nodes that would fail the AVL balance requirement.

Announcements

Exam I Topics:

ADTs

- Lists
- Stacks
- Queues
- Maps

Code Analysis

- Code Modeling
- Big O / Asymptotic Analysis
- Case Analysis
- Recurrences
- Master Theorem

Data Structures

- Arrays
- Linked Lists
- Hash Tables
- Binary Search Trees
- AVLs
- LLRBs
- Tries

Review: Dictionaries

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

- Why are we so obsessed with Dictionaries?

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

Operation		ArrayList	LinkedList	HashTable	BST	AVLTree
put(key,value)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)	Θ(1)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(log n)
get(key)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)	Θ(1)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(log n)
remove(key)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)	Θ(log n)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(log n)

Design Decisions

Before coding can begin engineers must carefully consider the design of their code will organize and manage data

Things to consider:

- What functionality is needed?
  - What operations need to be supported?
  - Which operations should be prioritized?
- What type of data will you have?
  - What are the relationships within the data?
  - How much data will you have?
  - Will your data set grow?
  - Will your data set shrink?
- How do you think things will play out?
  - How likely are best cases?
  - How likely are worst cases?

Example: Class Gradebook

You have been asked to create a new system for organizing students in a course and their accompanying grades

What functionality is needed?

- What operations need to be supported?
  - Add students to course
  - ➡ Add grade to student's record
  - Update grade already in student's record
  - Remove student from course
  - Check if student is in course
  - ➡ Find specific grade for student

Which operations should be prioritized?

What type of data will you have?

- What are the relationships within the data?
  - Organize students by name, keep grades in time order…
- How much data will you have?
  - A couple hundred students, < 20 grades per student
- Will your data set grow?
- Will your data set shrink?

A lot at the beginning,
Not much after that

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?
  - Lots of add and drops?
  - Lots of grade updates?
  - Students with similar identifiers?

Example: Class Gradebook

What data should we use to identify students? (keys)

How should we store each student’s grades? (values)

Which data structure is the best fit to store students and their grades?

Practice: Music Storage

You have been asked to create a new system for organizing songs in a music service. For each song you need to store the artist and how many plays that song has.

What functionality is needed?

- What operations need to be supported?
- Which operations should be prioritized?

Update number of plays for a song
Add a new song to an artist's collection
Add a new artist and their songs to the service
Find an artist's most popular song
Find service's most popular artist
more…

What type of data will you have?

- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

Artists need to be associated with their songs, songs need t be associated with their play counts
Play counts will get updated a lot
New songs will get added regularly

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?

Some artists and songs will need to be accessed a lot more than others
Artist and song names can be very similar

Practice: Music Storage

How should we store songs and their play counts?

How should we store artists with their associated songs?

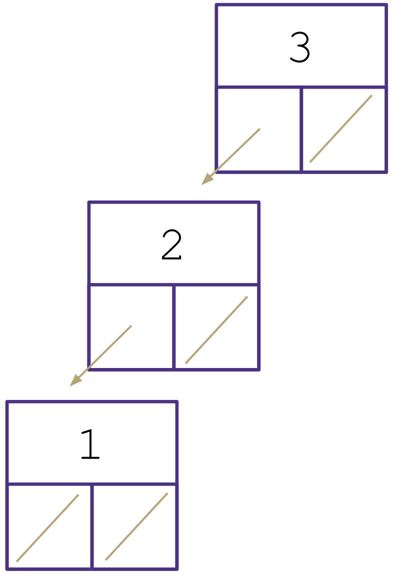
Questions?

AVLs

Two AVL Cases

Line Case

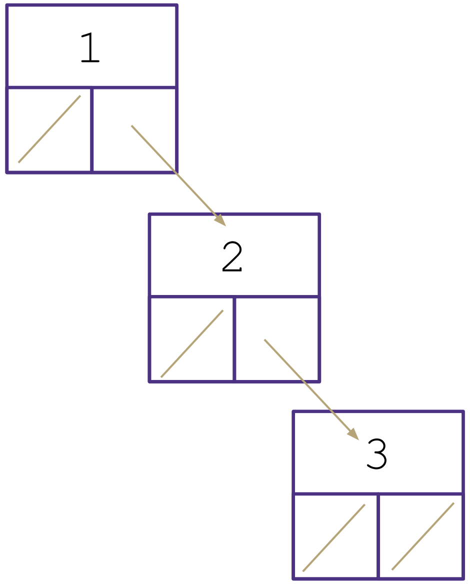
Solve with **1** rotation



Rotate Right

Parent’s left becomes child’s right

Child’s right becomes its parent



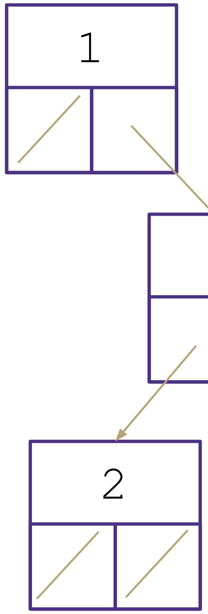
Rotate Left

Parent’s right becomes child’s left

Child’s left becomes its parent

Kink Case

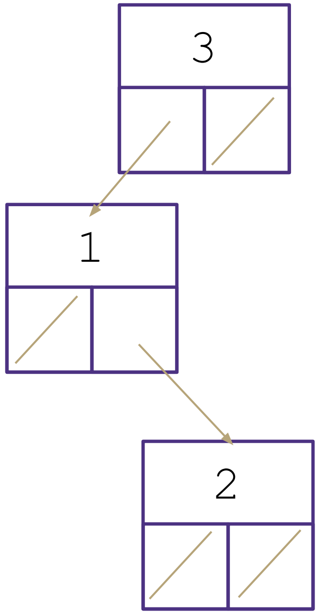
Solve with **2** rotations



Right Kink Resolution

Rotate subtree left

Rotate root tree right



Left Kink Resolution

Rotate subtree right

Rotate root tree left

How Long Does Rebalancing Take?

- Assume we store in each node the height of its subtree.
  - How do we find an unbalanced node?
  - Just go back up the tree from where we inserted.
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node.
  - A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion
  - log(n) time to traverse to a leaf of the tree
  - log(n) time to find the imbalanced node
  - constant time to do the rotation(s)
  - **Theta(log(n)) time for put** (the worst case for all interesting + common AVL methods (get/containsKey/put is logarithmic time)

AVL `insert()`: Approach

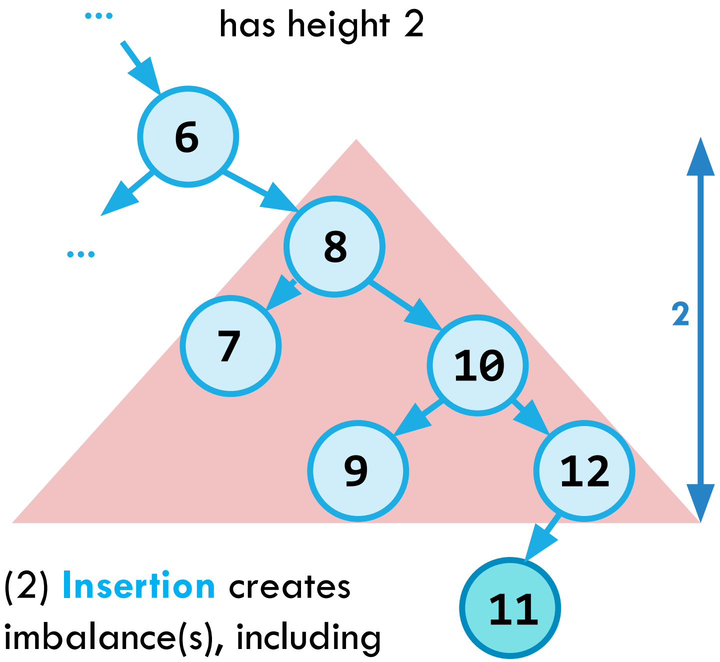
Our overall algorithm:

1. Insert the new node as in a BST (a new leaf)
2. For each node *on the path from the root to the new leaf*:
   - The insertion may (or may not) have changed the node's height
   - Detect height imbalance and perform a *rotation* to restore balance

Facts that make this easier:

- Imbalances can only occur along the path from the new leaf to the root
- We only have to address the lowest unbalanced node
- Applying a rotation (or double rotation), restores the height of the subtree before the insertion -- when everything was balanced!
- Therefore, we need ***at most one rebalancing operation***

(1) Originally, whole tree balanced, and this subtree has height 2



(2) Insertion creates imbalance(s), including the subtree (8 is lowest unbalanced node)

(3) Since the rotation on 8 will restore the subtree to height 2, whole tree balanced again!

AVL insert() code

```
Node insertNode(int key, Node node) {

   node = super.insertNode(key, node);

   updateHeight(node);

   return rebalance(node);

}
```

```
private void updateHeight(Node node) {
   int leftChildHeight = height(node.left);
   int rightChildHeight = height(node.right);
   node.height = max(leftChildHeight, rightChildHeight) + 1;
}
```

```
public class Node {
   int data;
   Node left;
   Node right;
   int height;

   public Node(int data) {
      this.data = data;
   }
}
```

```
private Node rebalance(Node node) {
   int balanceFactor = balanceFactor(node);

   // Left-heavy?
   if (balanceFactor < -1) {
      if (balanceFactor(node.left) <= 0) {    // Case 1
         // Rotate right
         node = rotateRight(node);
      } else {                                // Case 2
         // Rotate left-right
         node.left = rotateLeft(node.left);
         node = rotateRight(node);
      }
   }

   // Right-heavy?
   if (balanceFactor > 1) {
      if (balanceFactor(node.right) >= 0) {    // Case 3
         // Rotate left
         node = rotateLeft(node);
      } else {                                 // Case 4
         // Rotate right-left
         node.right = rotateRight(node.right);
         node = rotateLeft(node);
      }
   }
   return node;
}
```

AVL `rotate()` code

```
private Node rotateLeft(Node node) {
  Node rightChild = node.right;
  node.right = rightChild.left;
  rightChild.left = node;

  updateHeight(node);
  updateHeight(rightChild);
  return rightChild;
}
```

```
private Node rotateRight(Node node) {
  Node leftChild = node.left;
  node.left = leftChild.right;
  leftChild.right = node;

  updateHeight(node);
  updateHeight(leftChild);
  return leftChild;
}
```

AVL `delete()`

- Unfortunately, deletions in an AVL tree are more complicated
- There's a similar set of rotations that let you rebalance an AVL tree after deleting an element
    - Beyond the scope of this class
    - You can research on your own if you're curious!
- In the worst case, takes Θ(log *n*) time to rebalance after a deletion
    - But finding the node to delete is also Θ(log *n*), and Θ(2log *n*) is just a constant factor. Asymptotically the same time

- We won't ask you to perform an AVL deletion

AVL Trees

PROS

- All operations on an AVL Tree have a logarithmic worst case
  - Because these trees are always balanced!
- The act of rebalancing adds no more than a constant factor to insert and delete
- Asymptotically, just better than a normal BST!

CONS

- Relatively difficult to program and debug (so many moving parts during a rotation)
- Additional space for the height field
- Though asymptotically faster, rebalancing does take some time
  - Depends how important every little bit of performance is to you

More self balancing techniques

AVLs use rotations to maintain balance

- Balance maintains O(logn) performance

AVL rotations are very complex to implement

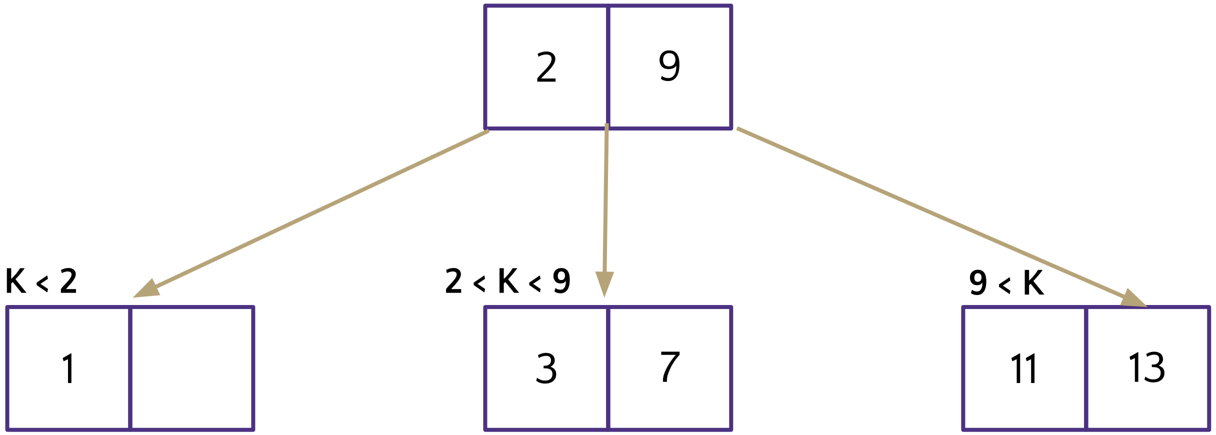
Other ways to maintain balance

- Condense multiple data points into a single node
- Two types of connections: red or black

2-3 Trees

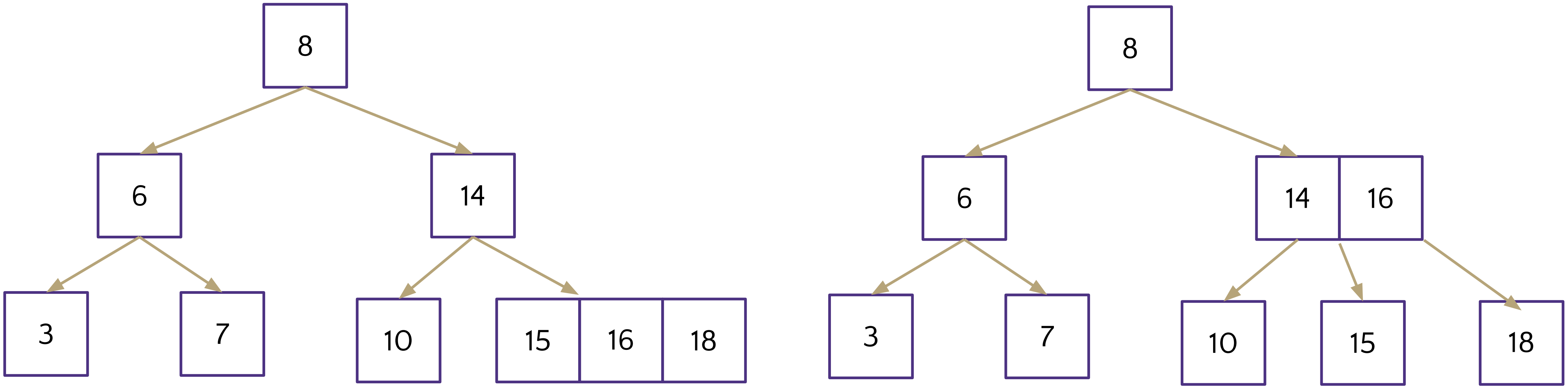
Properties:

- 2-nodes have 2 children and store 1 value
- 3-nodes have 3 children and store 2 values
- Data is stored in Binary Search order
- Tree is height balanced



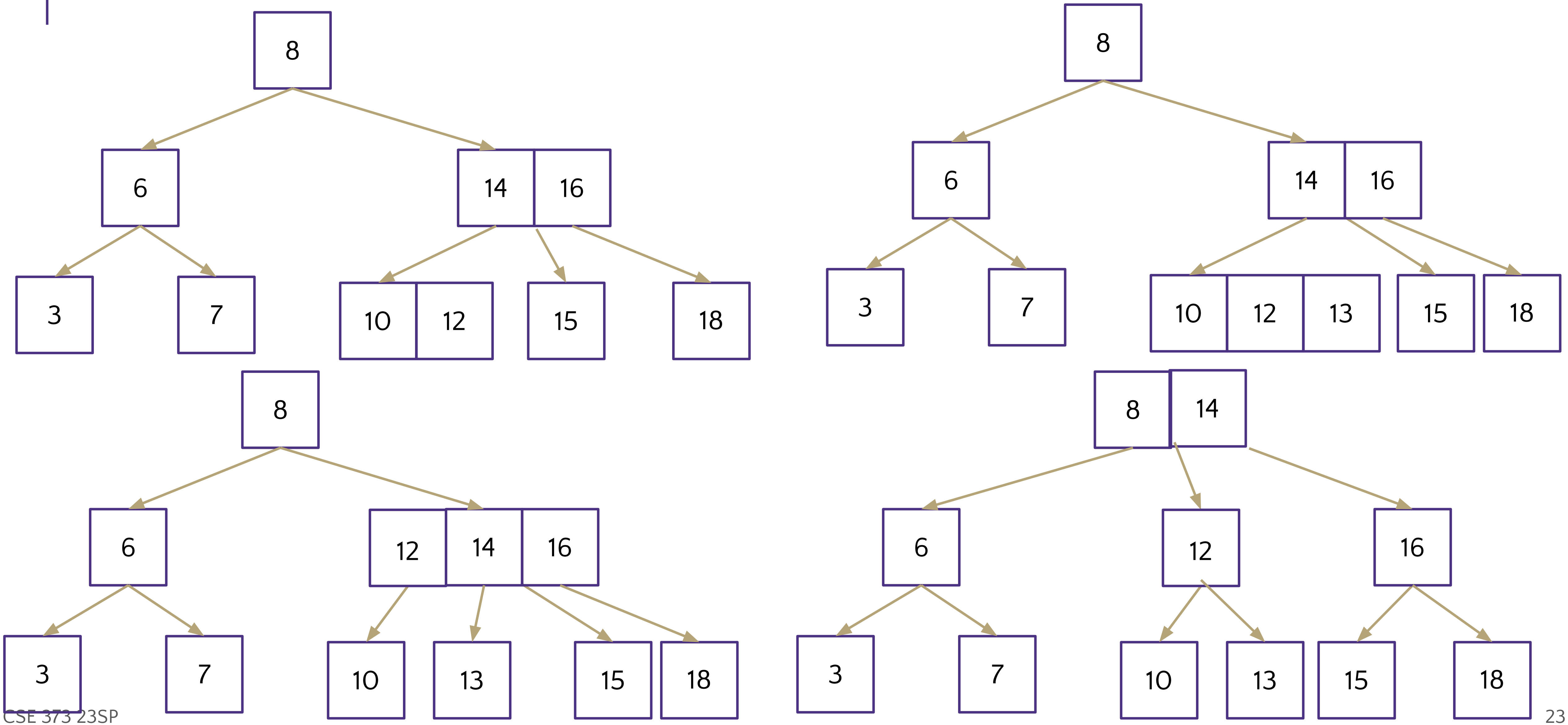
2-3 Insertions

1. Insert value into leaf node maintaining BST
2. If node is full, shift middle value up to parent
3. Split leaf node to satisfy required number of children



2-3 Insertions

Insert 12 and 13 into the following 2-3 tree



2-3 Trees

PROS

- All operations on 2-3 Tree have a logarithmic worst case
  - Because these trees are always balanced!
- Maintaining balance doesn't require complex rotations
- Storing multiple values per node improves runtime constants because of memory locality

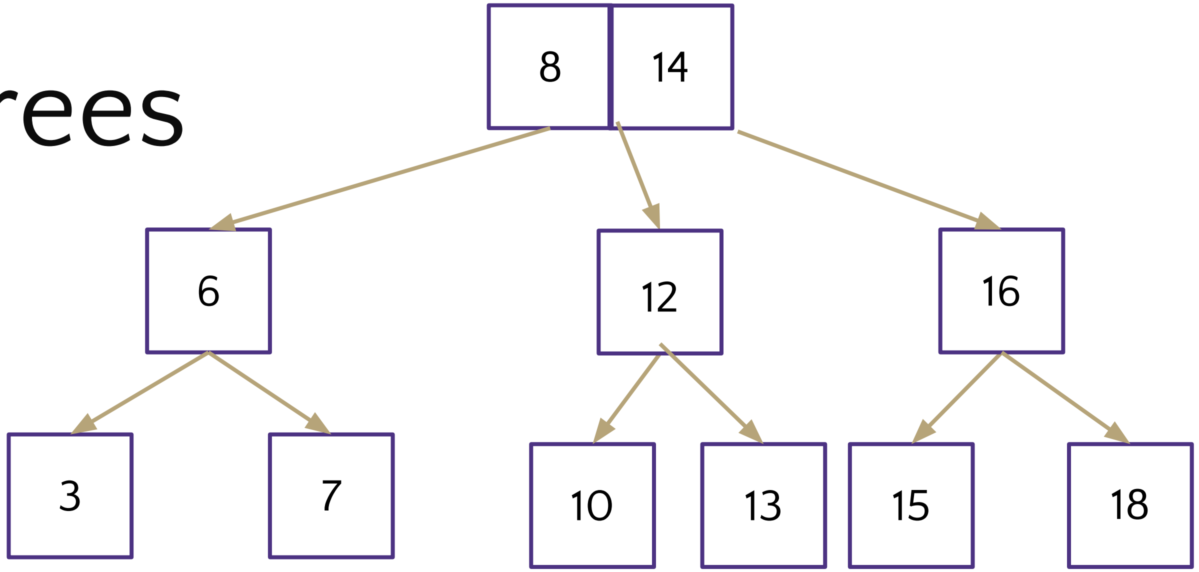
CONS

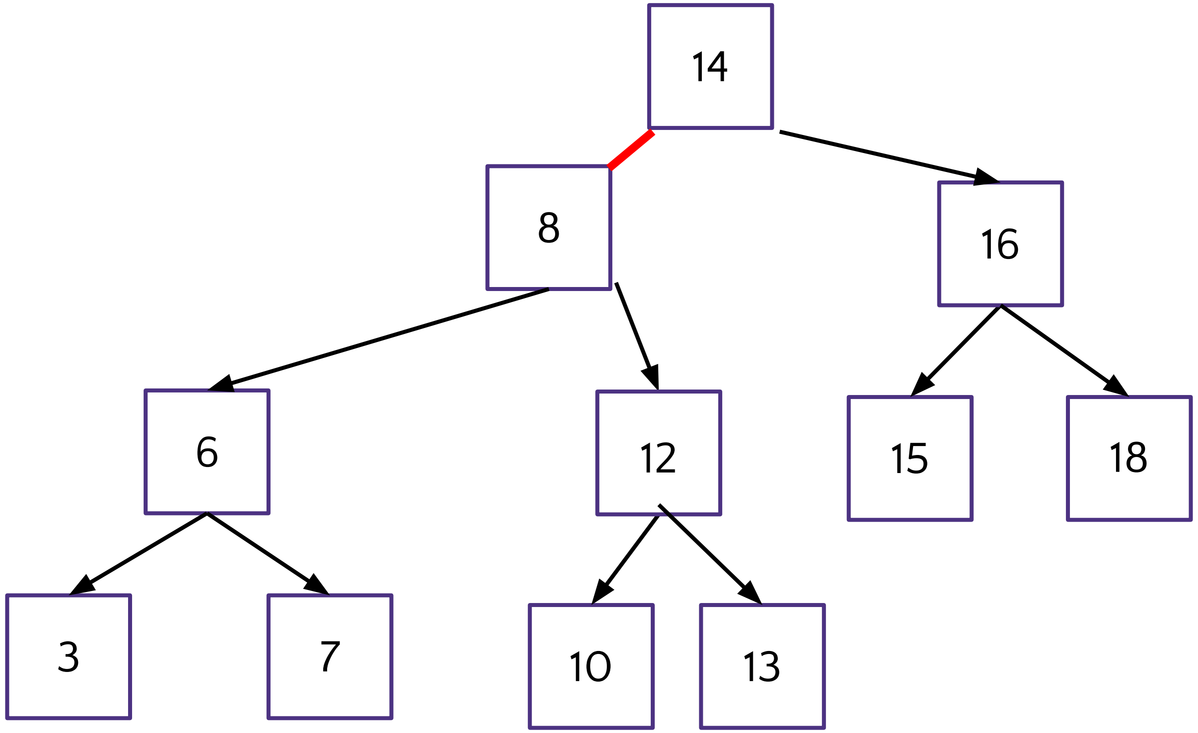
- No height triggered balancing means 2-3 trees stay a little less balanced than AVLs on average
- Multiple node types cause implementation complexity
  - Make all nodes 2 nodes and you have more unused space

Left Leaning Red Black Trees

A translation of 2 3 trees using nodes with only 1 value

- Red links connect two nodes that would exist within the same node in a 2-3 tree
- Black links are “standard” connections
- Red links are always on the left
- A “balanced” LLRB has the same number of black links to leaf
  - Red links don’t count towards path length





Your toolbox so far…

ADT

- List – flexibility, easy movement of elements within structure
- Stack – optimized for first in last out ordering
- Queue – optimized for first in first out ordering
- Dictionary (Map) – stores two pieces of data at each entry

<– It's all about data baby!

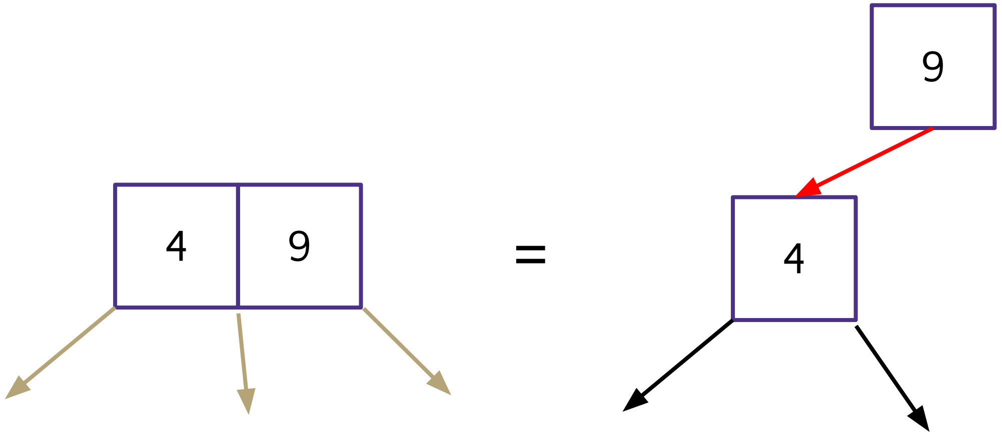
SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters

Data Structure Implementation

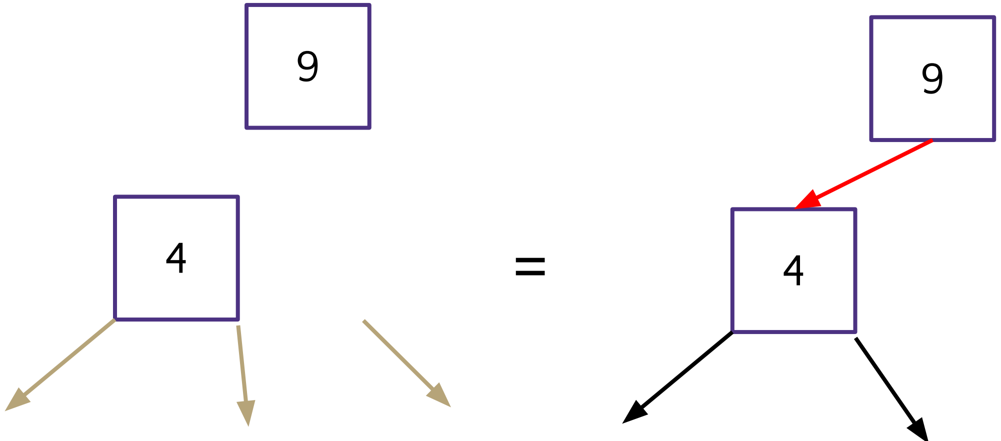
- Array – easy look up, hard to rearrange
- Linked Nodes – hard to look up, easy to rearrange
- Hash Table – constant time look up, no ordering of data
- BST – efficient look up, possibility of bad worst case
- AVL Tree – efficient look up, protects against bad worst case, hard to implement

“left leaning”

When you “split” the 3 nodes, turn them into a let leaning set



When you insert new nodes, add to leaf then do any appropriate rotations to ensure left lean



Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

- AVL tree
- Splay tree
- 2-3 tree
- AA tree
- Red-black tree
- Scapegoat tree
- Treap

(Not covered in this class, but several are in the textbook and all of them are online!)

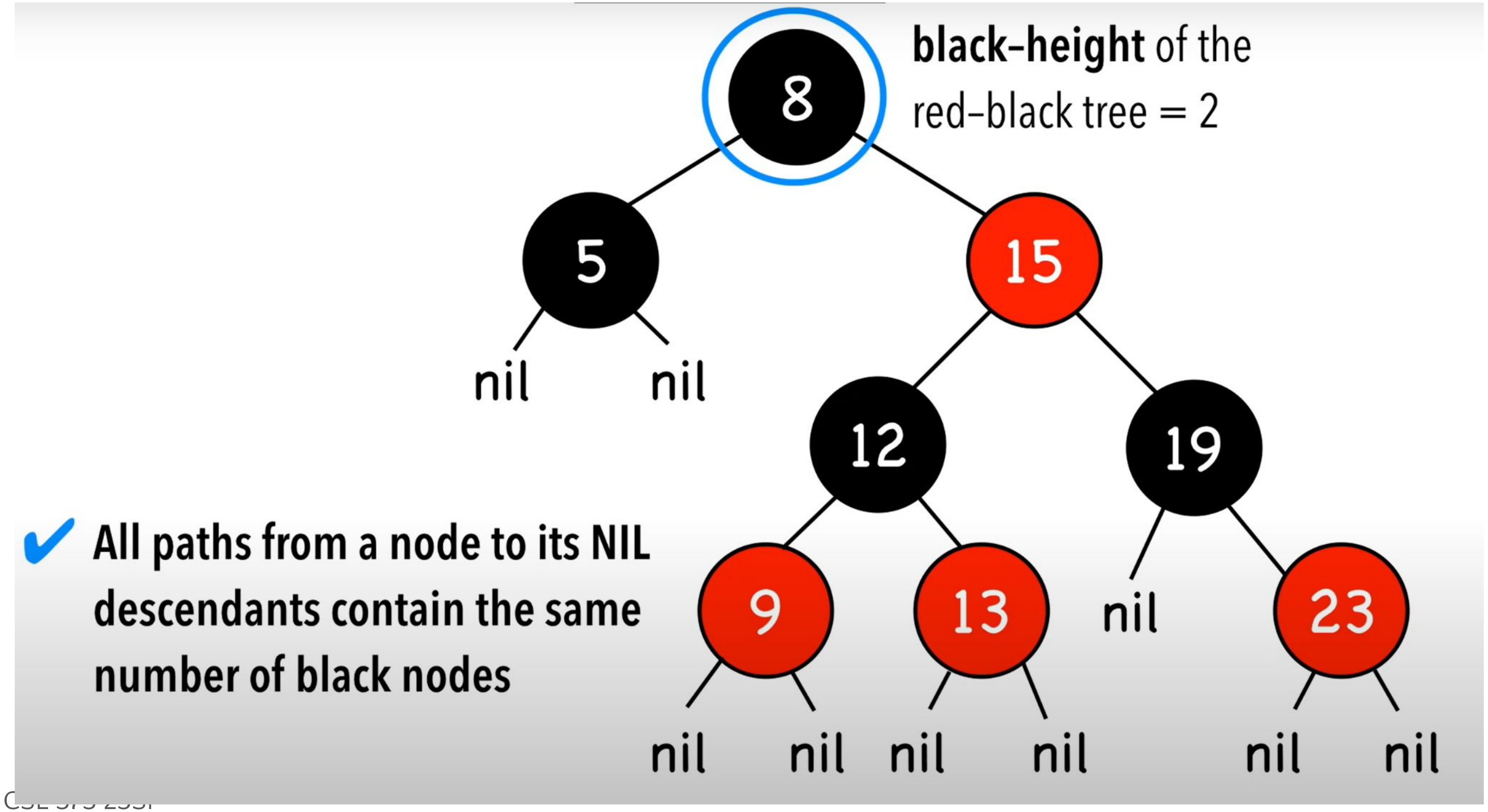
(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

Appendix

Red Black Requirements

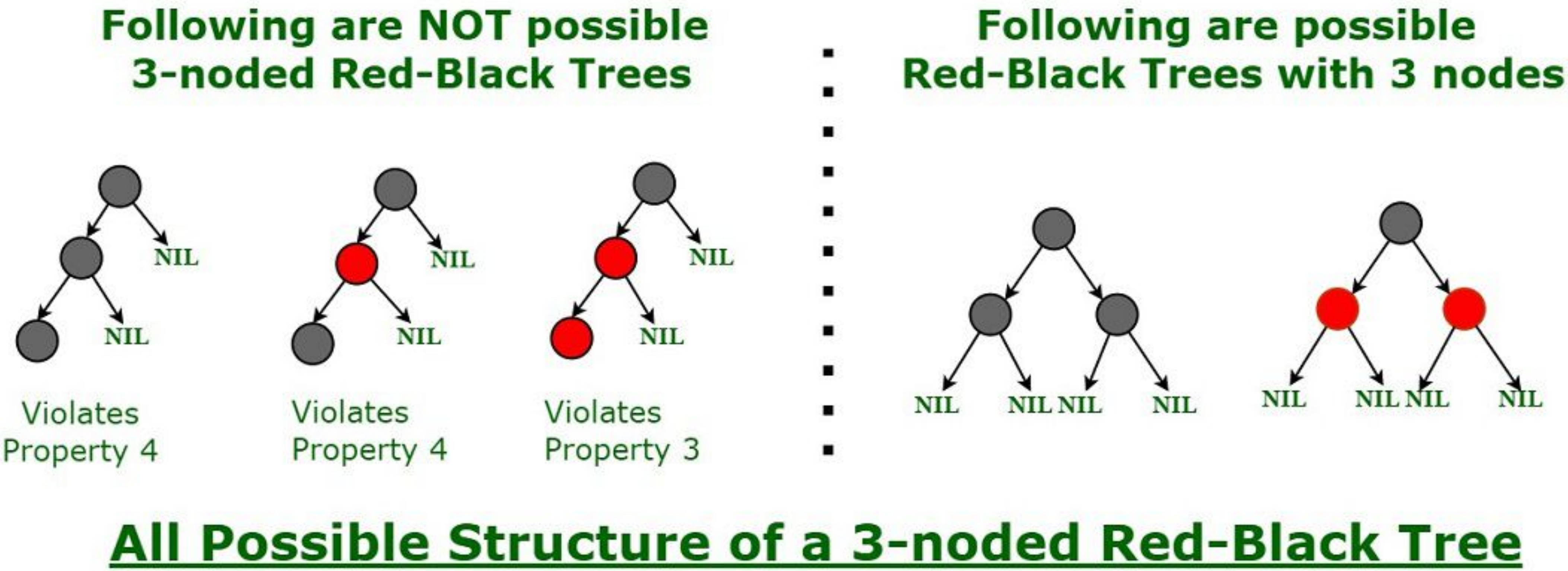
- BST Property
- All nodes have an extra field to mark them either “red” or “black”
- Overallroot is black
- If a node is red, it’s parent and children must be black
- All paths through the tree must have the same number of black nodes
  - Shortest path will be all black nodes
  - Longest path will alternate between black and red nodes

Valid Red Black tree



black-height of the red–black tree = 2

✔ All paths from a node to its NIL descendants contain the same number of black nodes



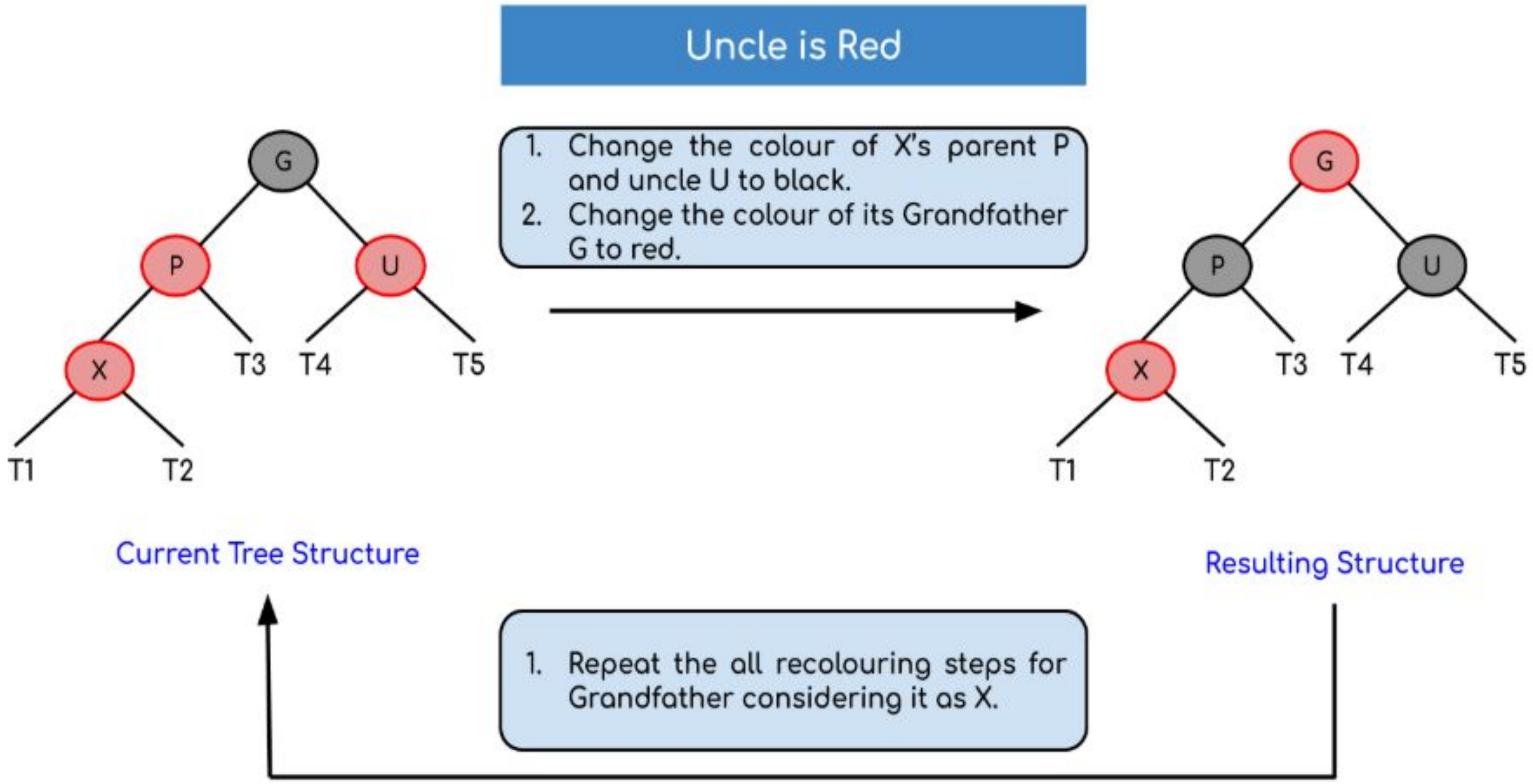
Following are NOT possible 3-noded Red-Black Trees

Following are possible Red-Black Trees with 3 nodes

All Possible Structure of a 3-noded Red-Black Tree

Red Black Tree Insertions

Recoloring



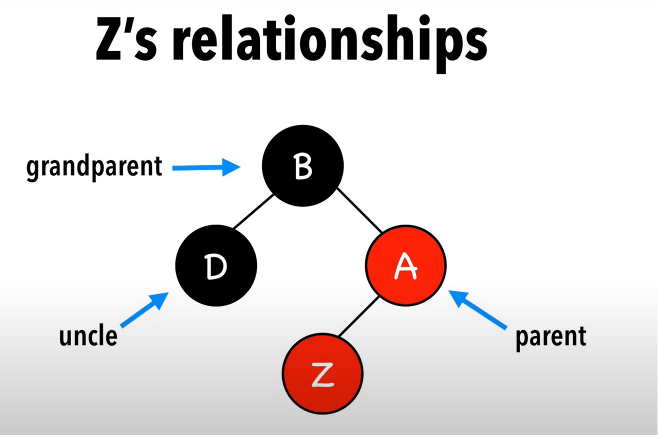
Uncle is Red

1. Change the colour of X's parent P and uncle U to black.
2. Change the colour of its Grandfather G to red.

Current Tree Structure

Resulting Structure

1. Repeat the all recolouring steps for Grandfather considering it as X.



Z's relationships

Red Black Insertions

1. Insert node and color it red
   a. This may break the root and leaves are black or red nodes must have black children properties but these are easy inariants to fix
   b. This wont break the length of paths must all have same number of black nodes property

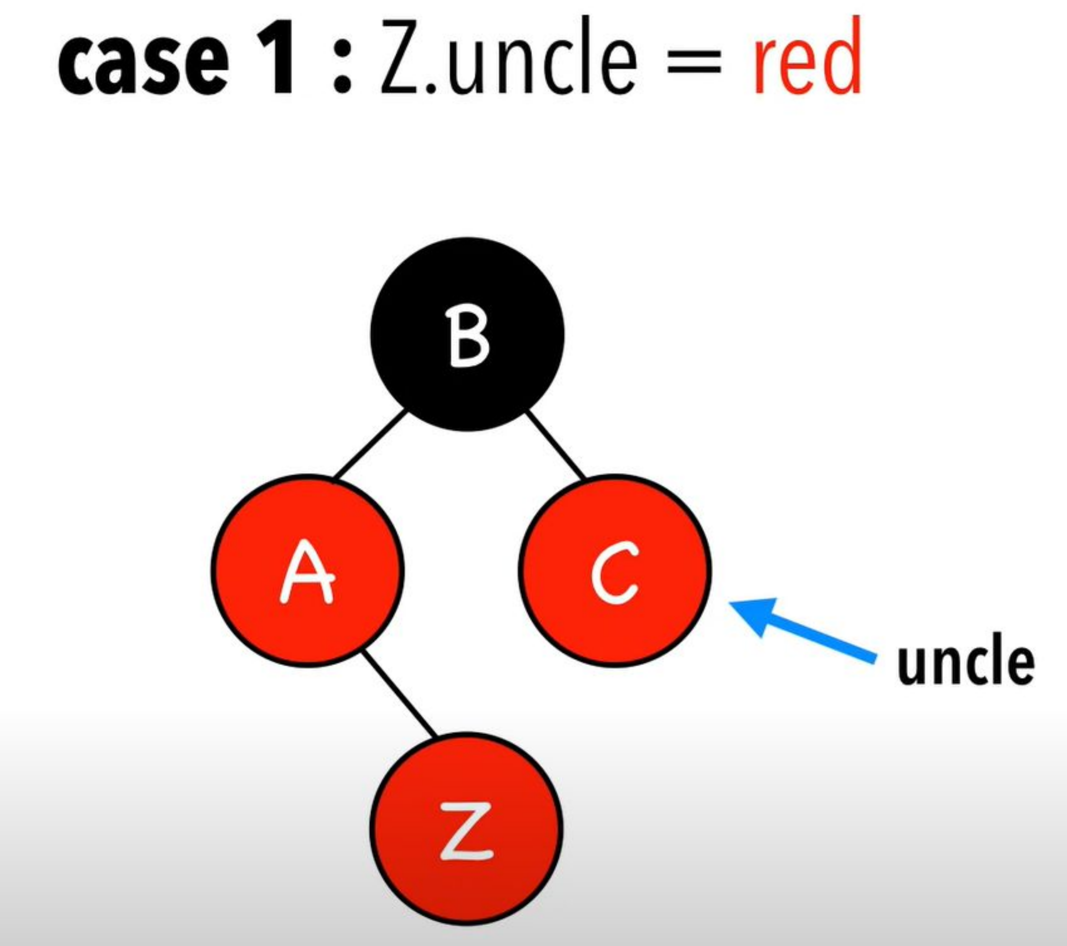
Insertion cases:

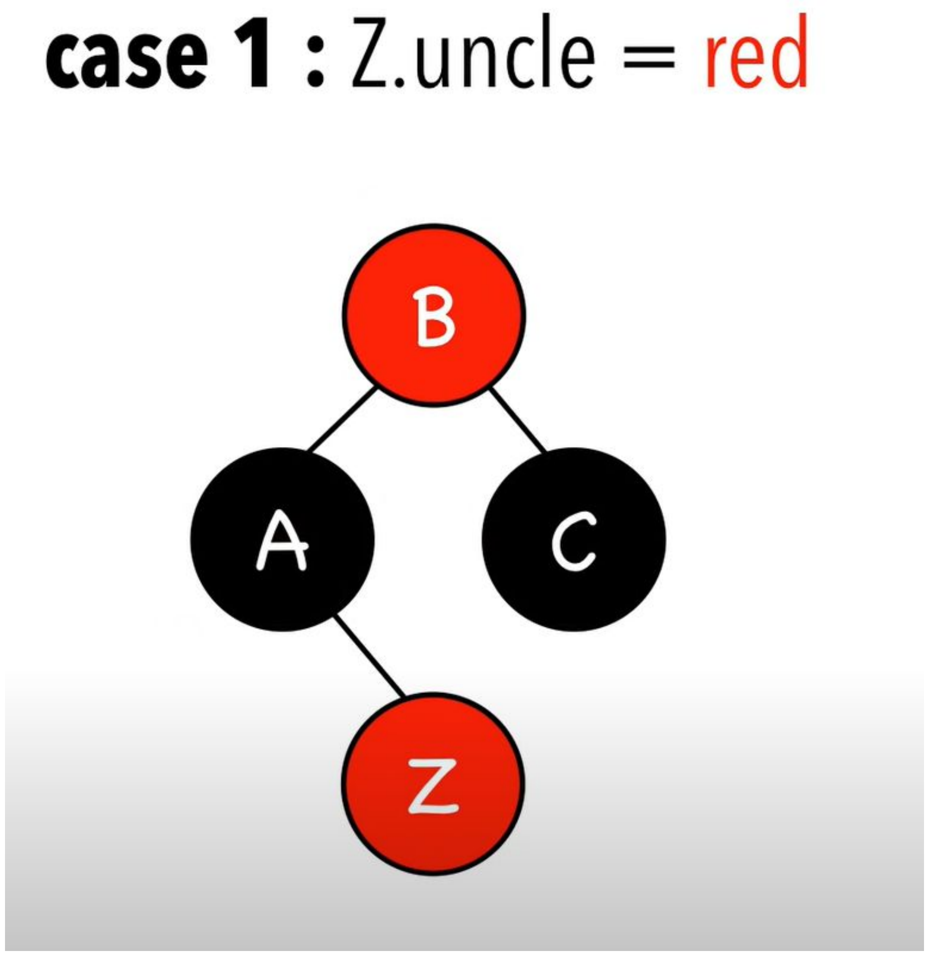
1. Node is the root
   a. Color node black
2. Node's uncle is red
   a. recolor
3. Node's uncle is black (Triangle)
   a. Rotate node's parent
4. Node's uncle is black (line)
   a. Rotate nodes' grandparent & recolor

Node’s uncle is red

What if the grandparent is the overallRoot?

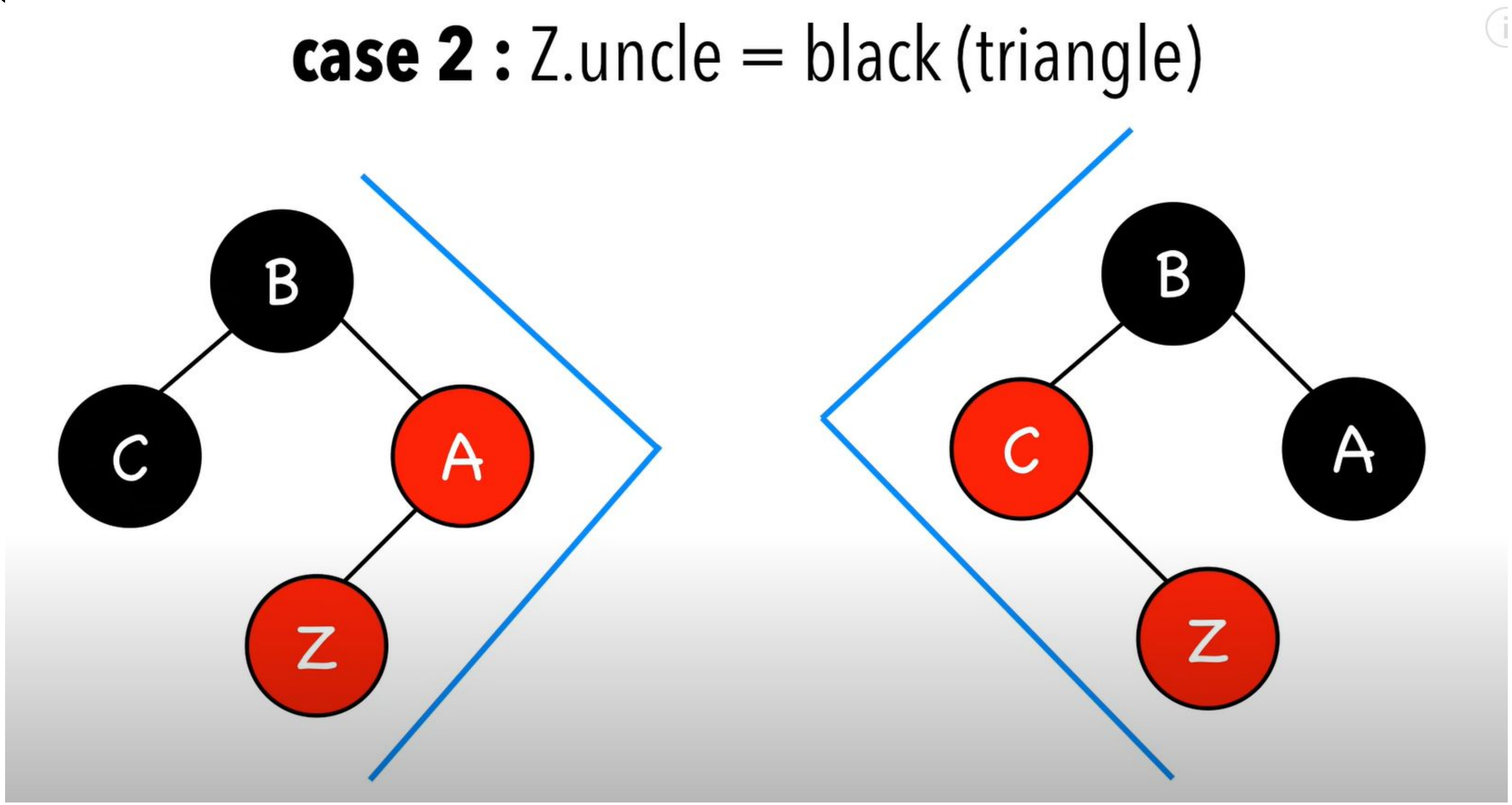
Recolor parent, uncle and grandparent





Uncle is black (triangle)

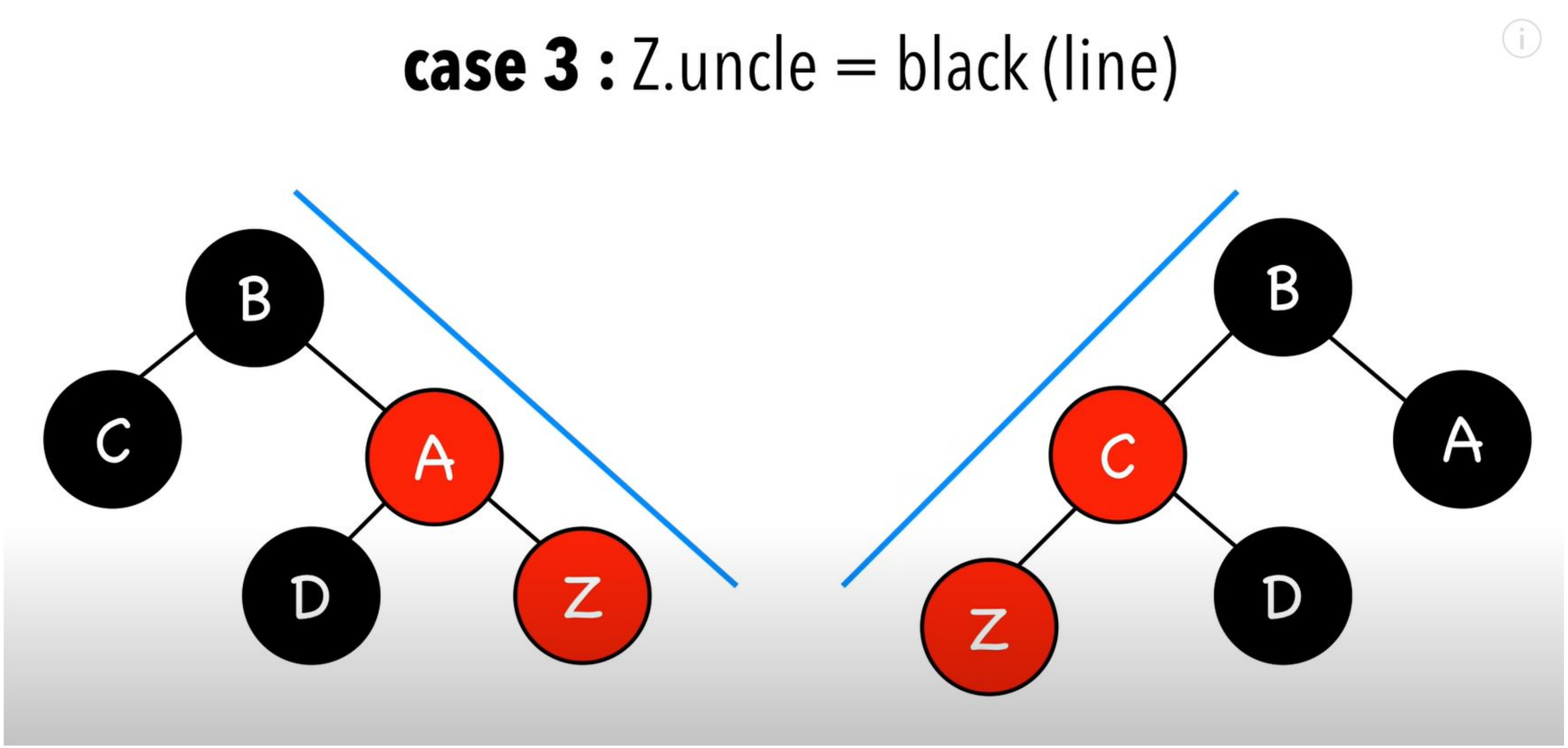
Rotate inserted Nodes parent in opposite direction of inserted node



case 2 : Z.uncle = black (triangle)

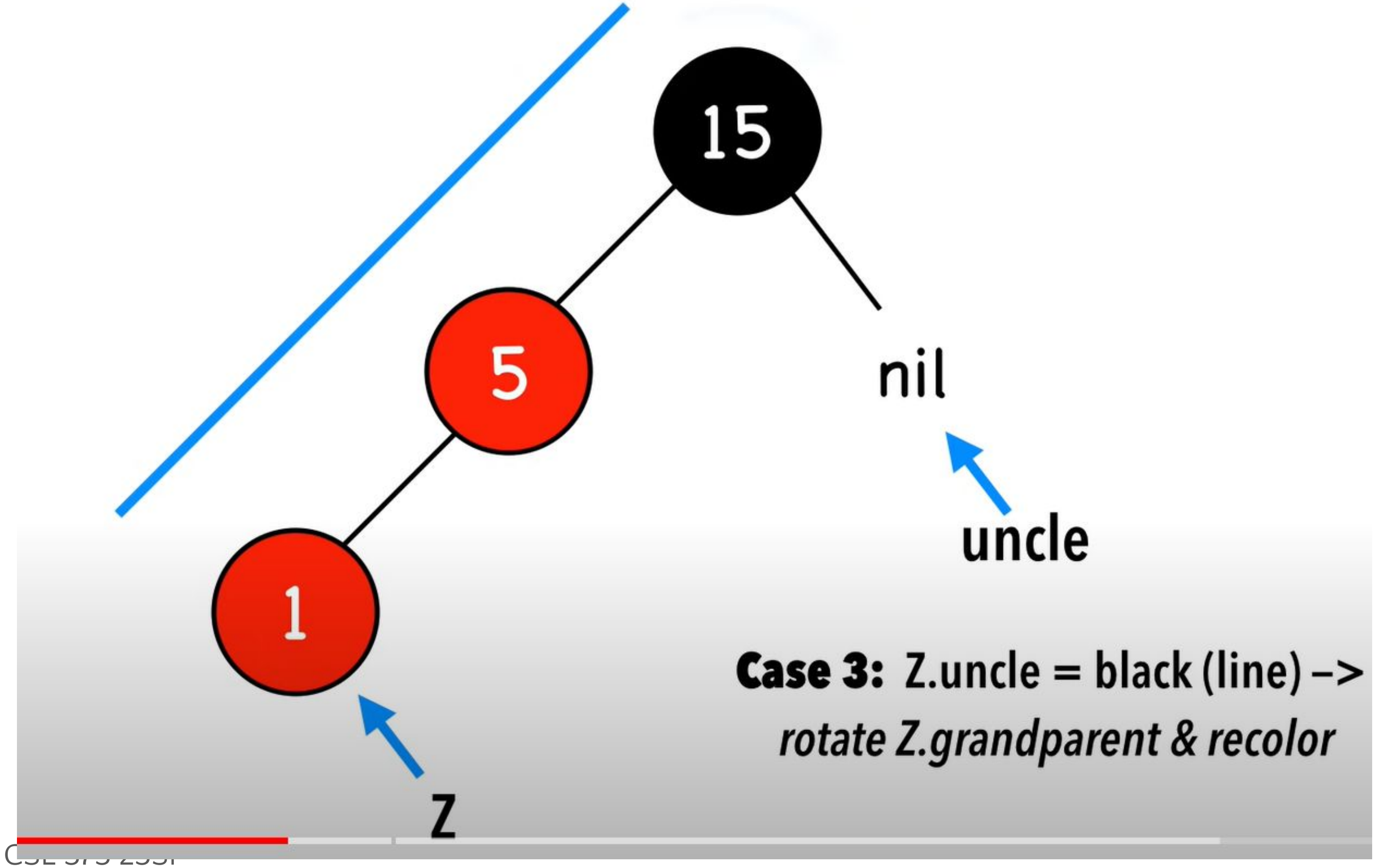
Uncle is black (line)

Rotate node's grandparent, then recolor

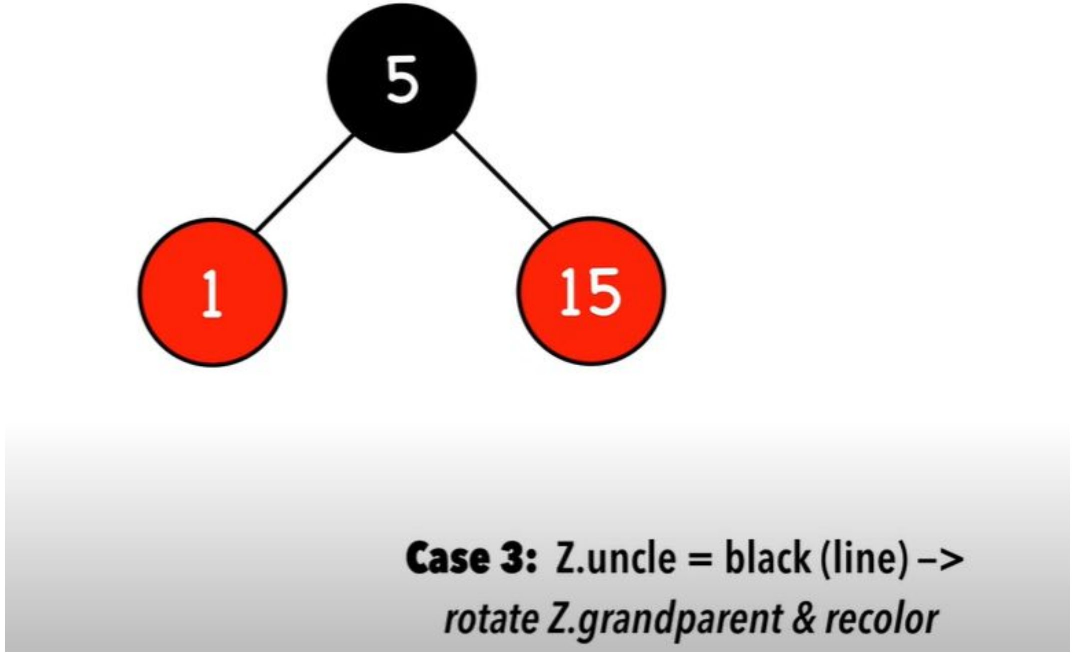


case 3 : Z.uncle = black (line)

Example



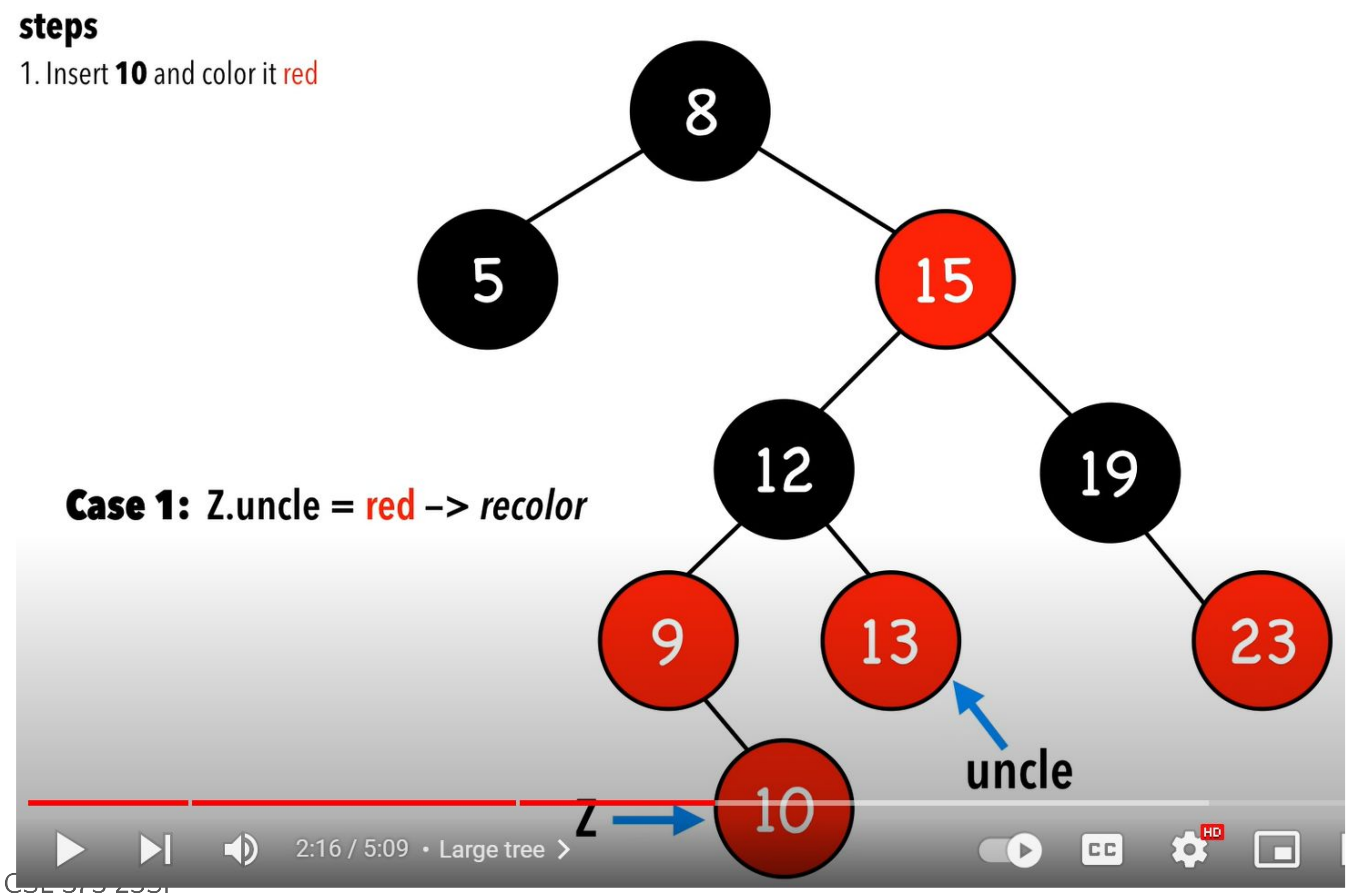
Case 3: Z.uncle = black (line) –>
rotate Z.grandparent & recolor



Case 3: Z.uncle = black (line) –>
rotate Z.grandparent & recolor

Example

https://www.youtube.com/watch?v=A3JZinzkMpk



steps

1. Insert **10** and color it red

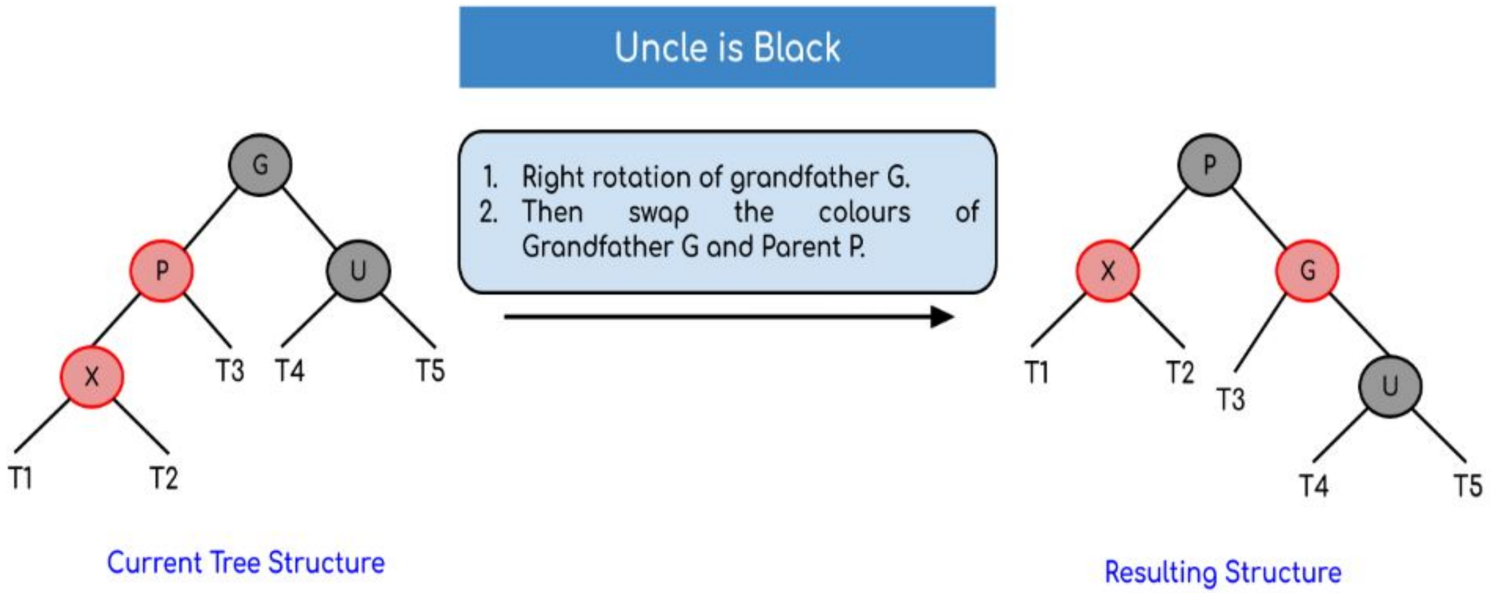
Case 1: Z.uncle = red -> *recolor*

Insertion

If parent and uncle of inserted node are red rotate

4 rotation cases (same as AVL)

- Left Left Case



Uncle is Black

1. Right rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.

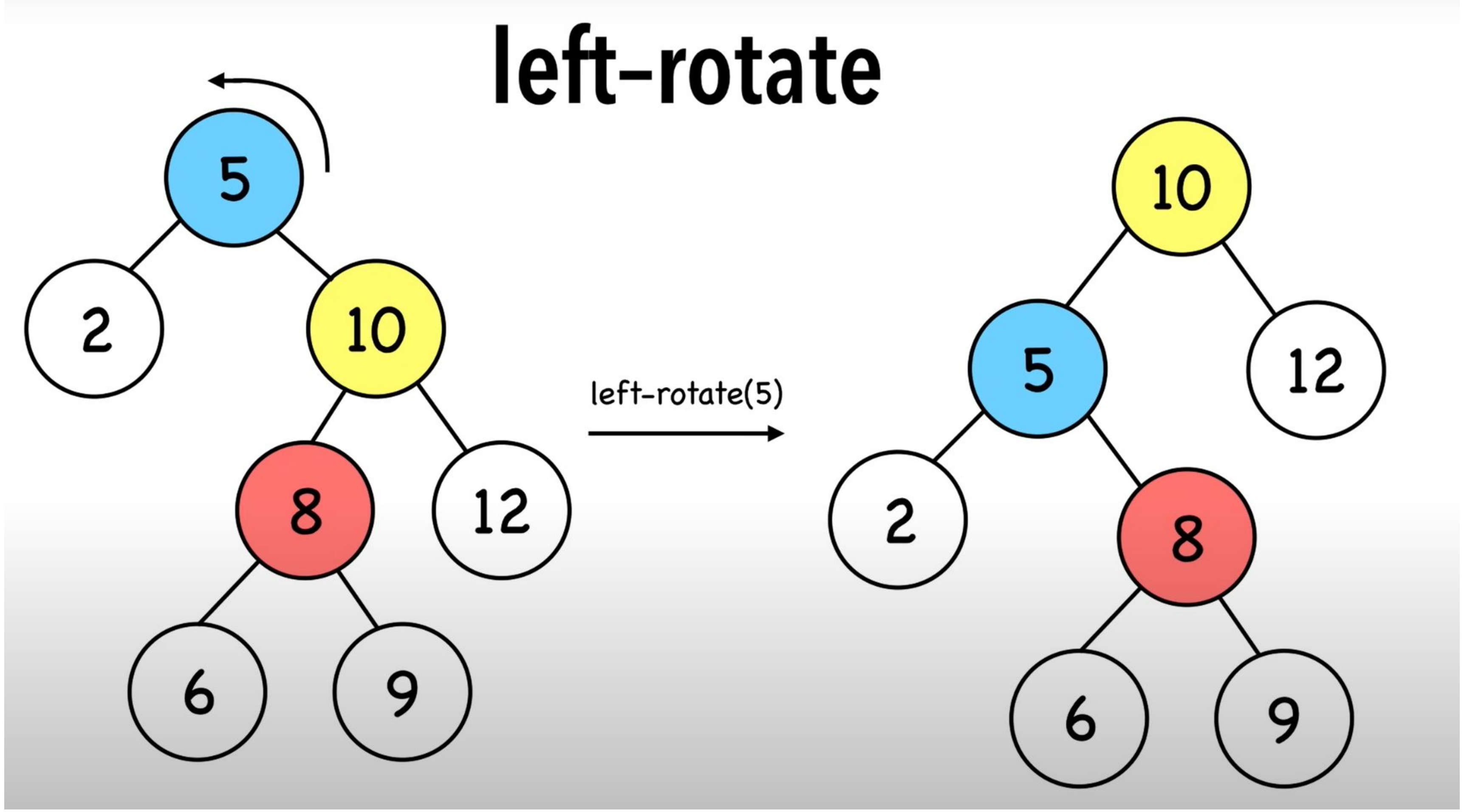
Current Tree Structure

Resulting Structure

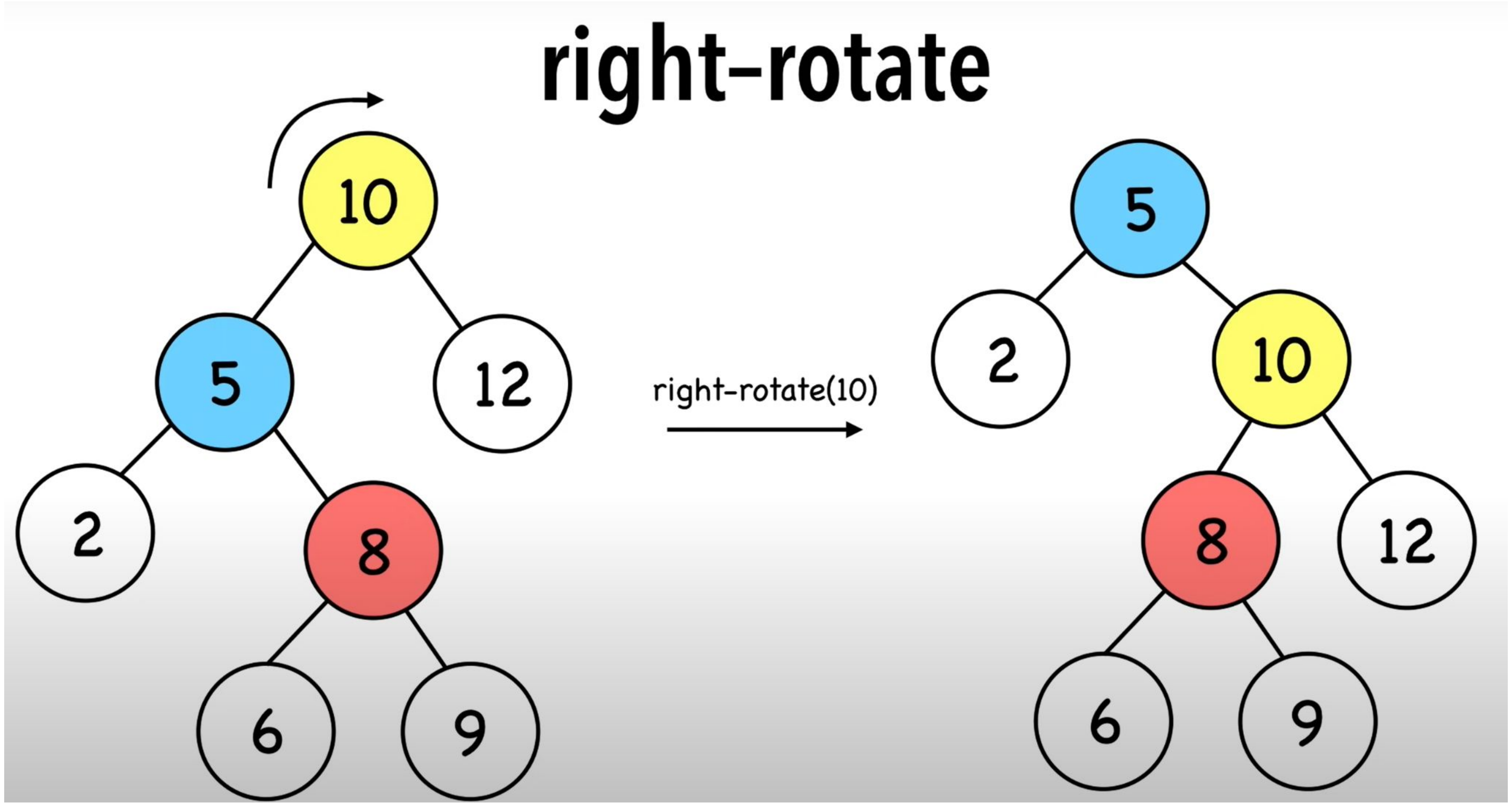
Left Right Case

1. Left rotation of Parent P.
2. Then apply LL rotation case.

Rotations



Rotations



AVL vs Red Black Trees

- AVLs maintain a more balanced tree
  - Insertions and deletions can trigger more rotations than Red Black Tree
-