Red-Black Trees

Red-Black Trees

- Invented by Guibas and Sedgewick in 1978.
  - Data structure of choice for implementing *maps* and *sets* in C++ Standard Template Library.
- Red-Black Trees are BSTs.
- Used to represent 2-3-4 Trees.
  - In a sense, BST are 2-3-4 Trees with only 2-nodes.
  - The 3-nodes and 4-nodes are "encoded" in the nodes
  - This encoding is represented in the node being either **RED** or **BLACK.**

Advantages of Red-Black Trees

- Since R-B Trees are BSTs, the standard search methods for BSTs work as-is.
- They correspond directly to 2-3-4 trees, so they are (mostly) always balanced.
- This means that searching, inserting and re-balancing are all **O(lg N)**.
- The insertion/re-balancing algorithm is fairly simple. However, coming up with the algorithm is not.

Properties of Red-Black Trees

- A R-B Tree is a BST, so it contains a link to both *left* and *right* children.
- Each node also contains a color code either **RED** or **BLACK**
- Additionally, it contains a pointer to it's *parent*
- *Note that "RED" and "BLACK" are arbitrary. The terms are simply tags to distinguish between the two types of nodes.*

```
enum COLOR { rbRED, rbBLACK };
struct RBNode
{
     RBNode *left;
     RBNode *right;
     RBNode *parent;
     COLOR color;
     void *item;
};
```

Properties of Red-Black Trees(2)

- Each node is marked as **RED** or **BLACK.**
- Newly inserted nodes are marked as **RED**.
- *NULL* nodes (empty children) are marked as **BLACK**.
- If a node is **RED**, then it's children must be **BLACK.**
  - *This means that two* **RED** *nodes are never adjacent on a path.*
- Every path from a anode to any of its leaves contains the same number of **BLACK** nodes.
- The root of the tree is **BLACK**. Technically, the root may be **RED**. But to keep the algorithm simple and ensure that everyone's trees look identical we'll require the root to be **BLACK.**

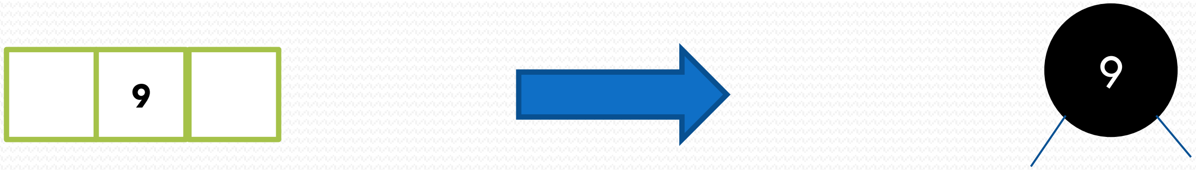
Properties of Red-Black Trees(3)

- Another way to state this is to focus on these two conditions:
  - The **RED** condition:
    - Each **RED** node has a **BLACK** parent.
  - The **BLACK** condition:
    - Each path from the root to every external node contains exactly the same number of **BLACK** nodes.

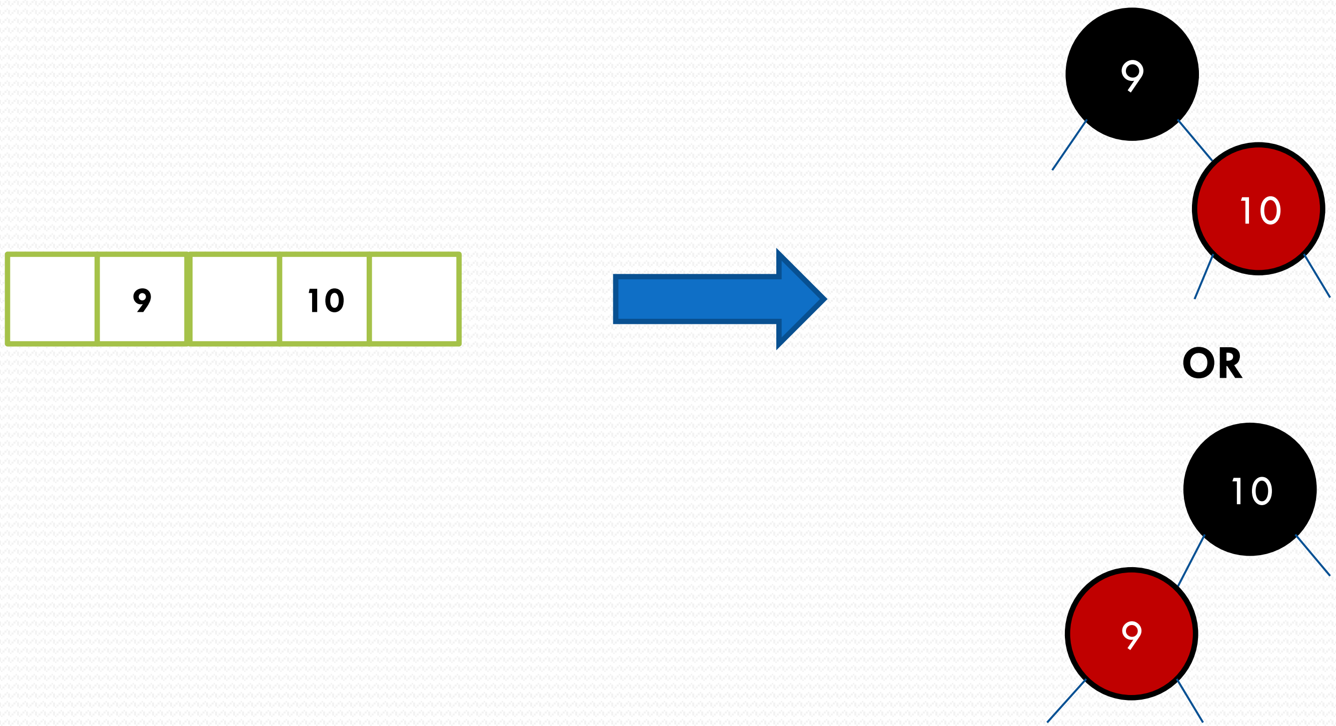
Mapping 2-3-4 Trees into RBTs

- Remember that Red-Black Trees are used to represent 2-3-4 trees in a BST form.
- It is possible to map any 2-3-4 Trees into a Red-Black Tree and vice versa.
- There are several situations:
    - 2 – nodes
    - 3 – nodes
    - 4 – nodes
    - 2-nodes connected to 3-nodes
    - 3-nodes connected to 4-nodes

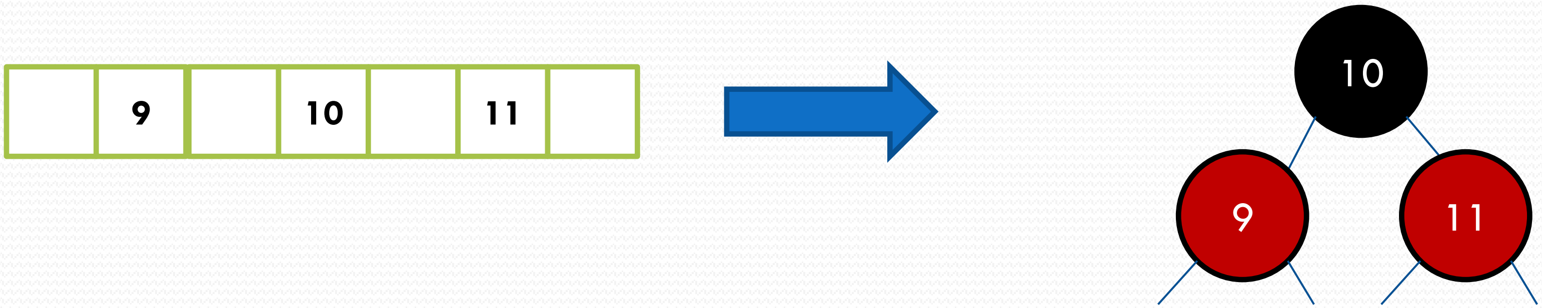
2-Node to RBT



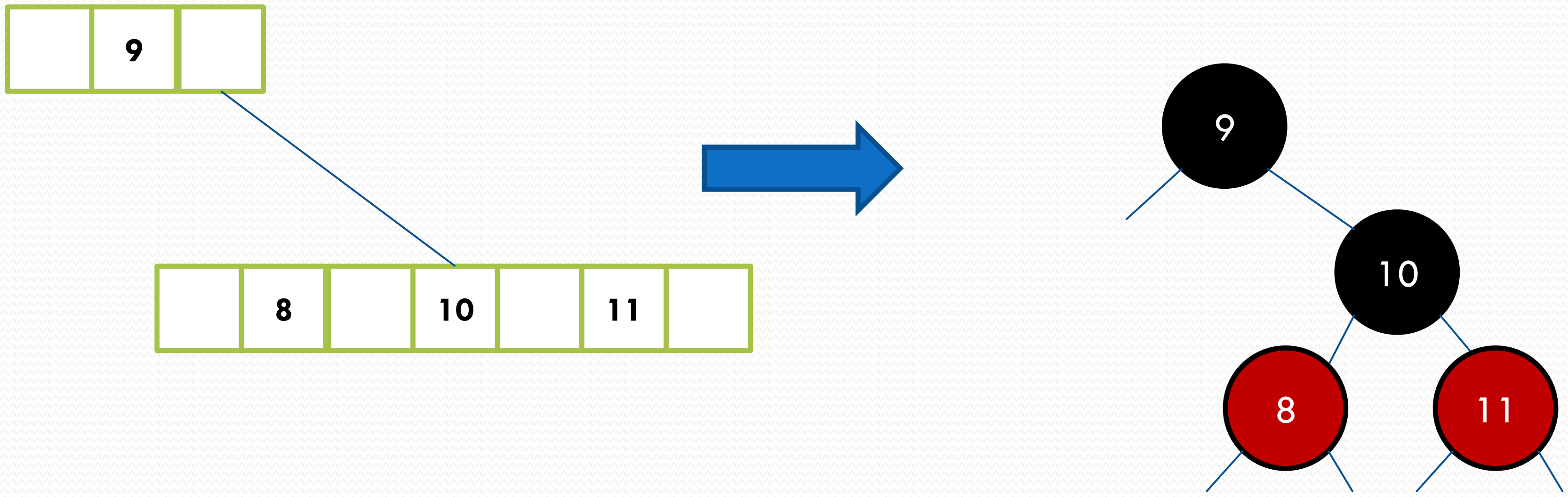
3-Node to RBT



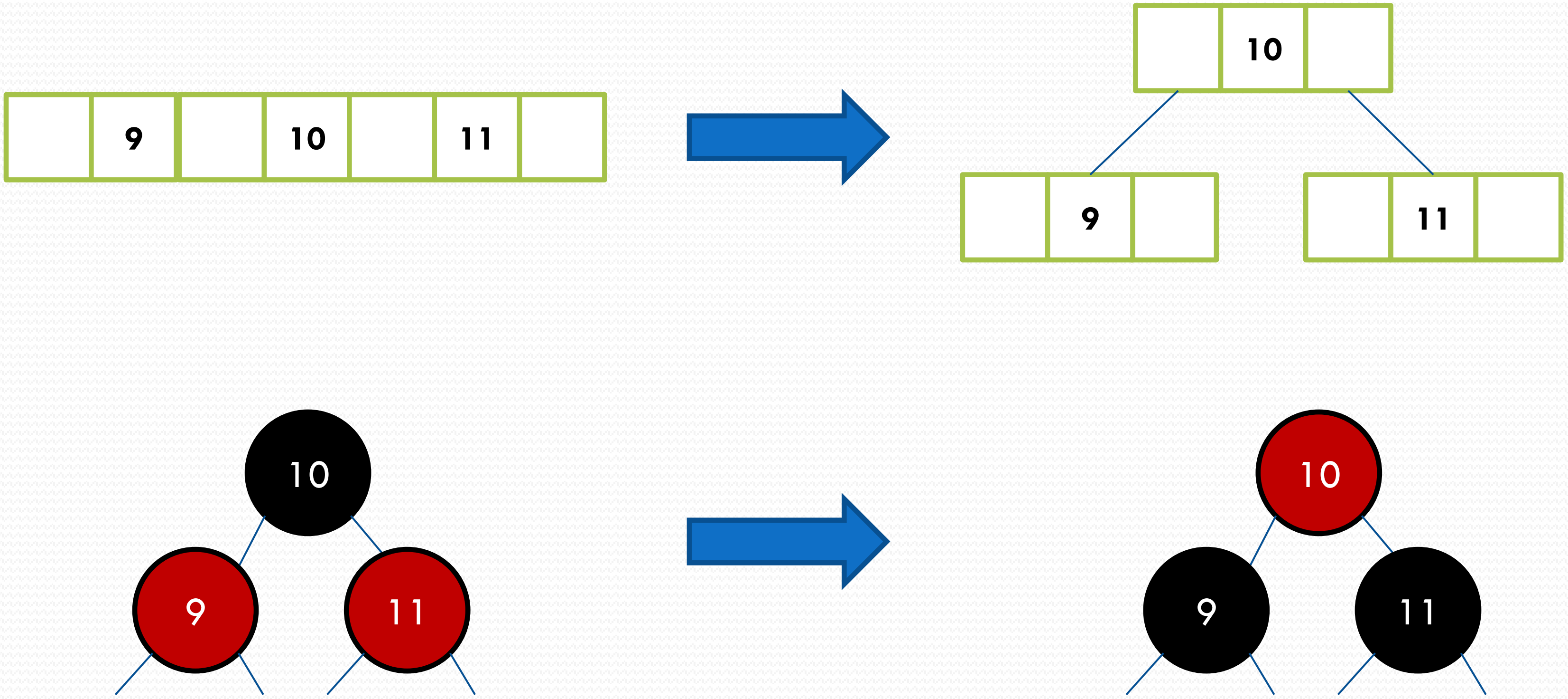
4-Node to RBT



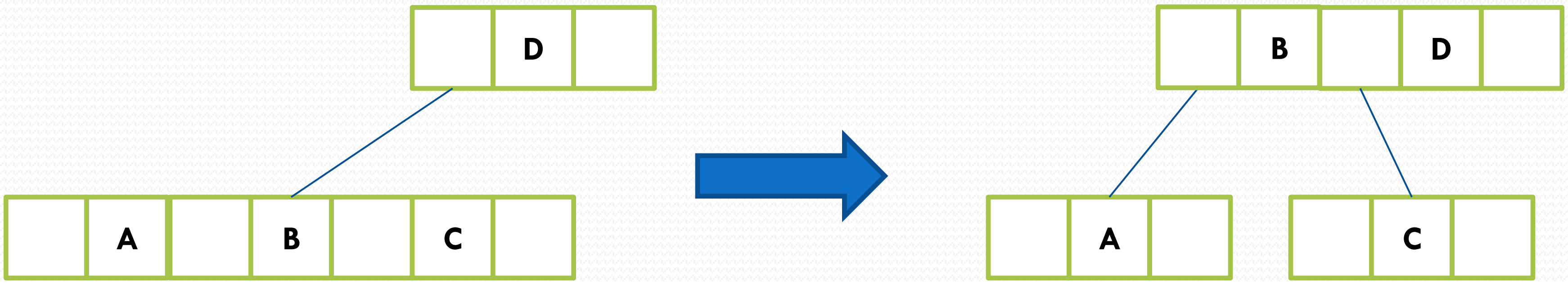
2-node connected to a 4-Node

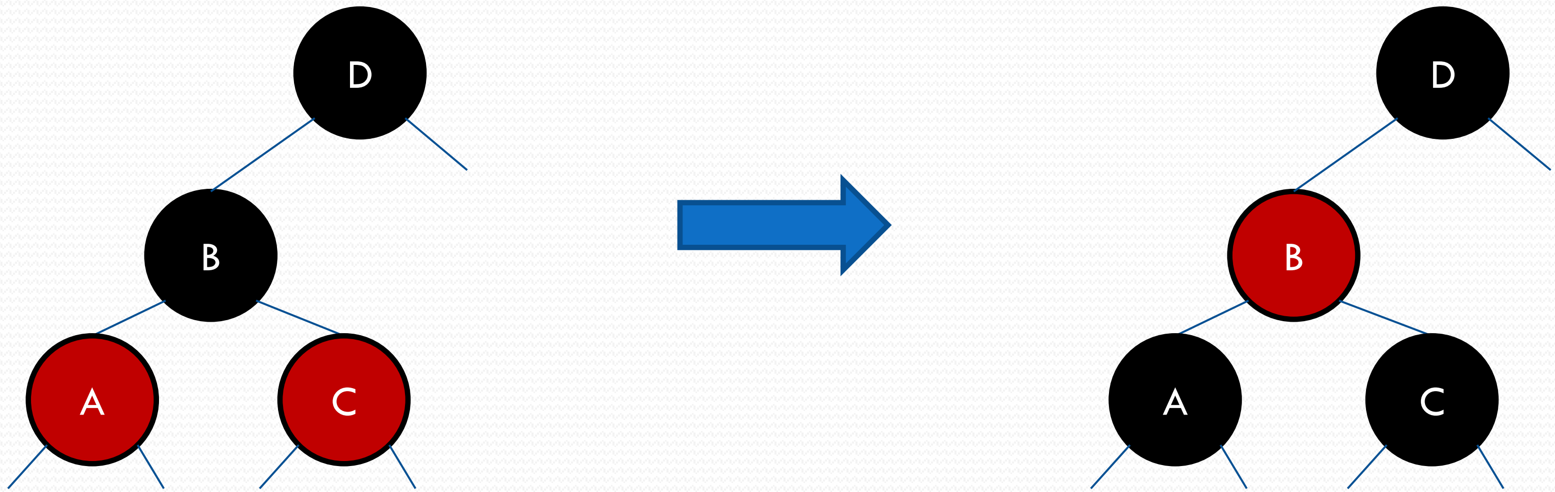


Splitting a 4-node:

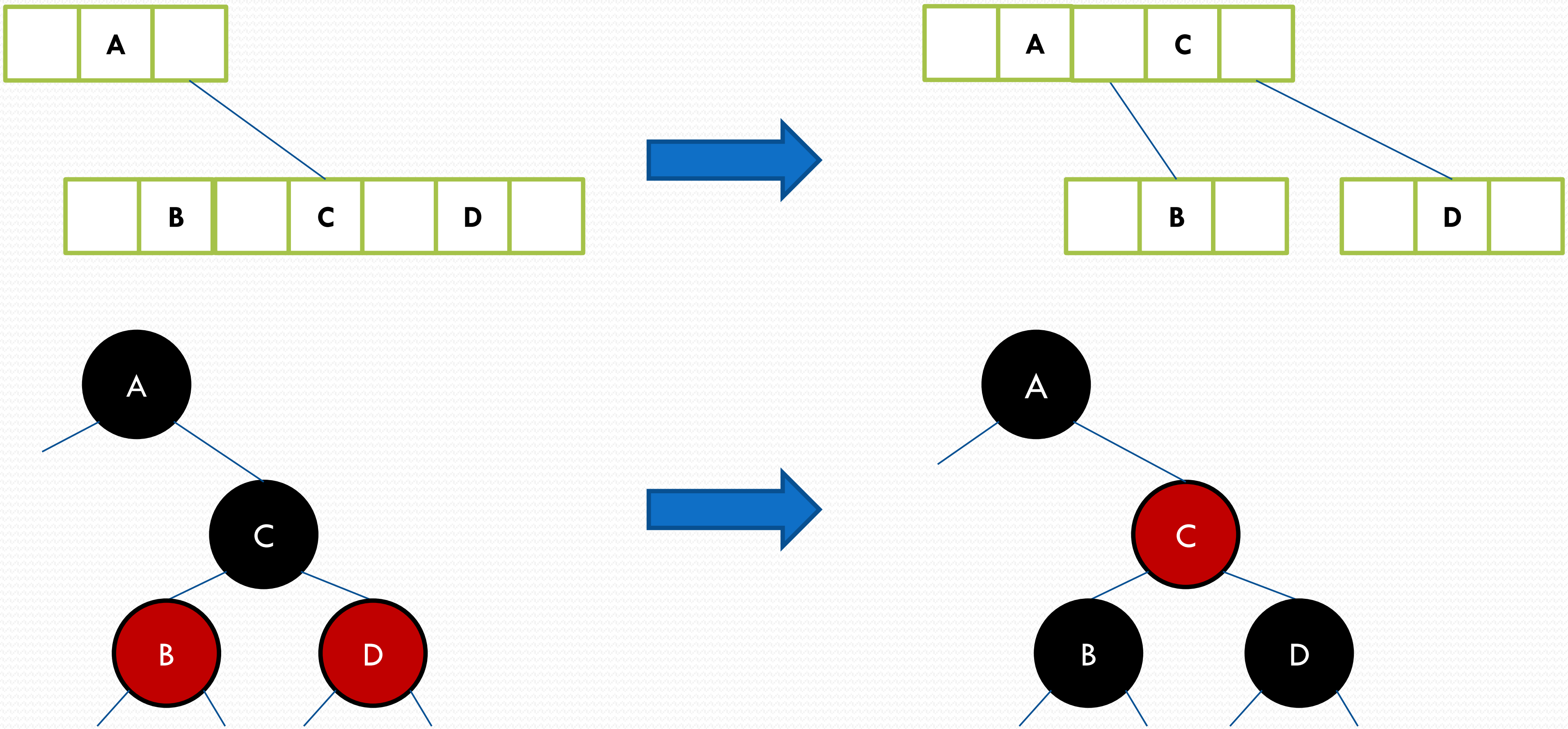


Splitting a 4-node connected to a 2-node (orientation #1):

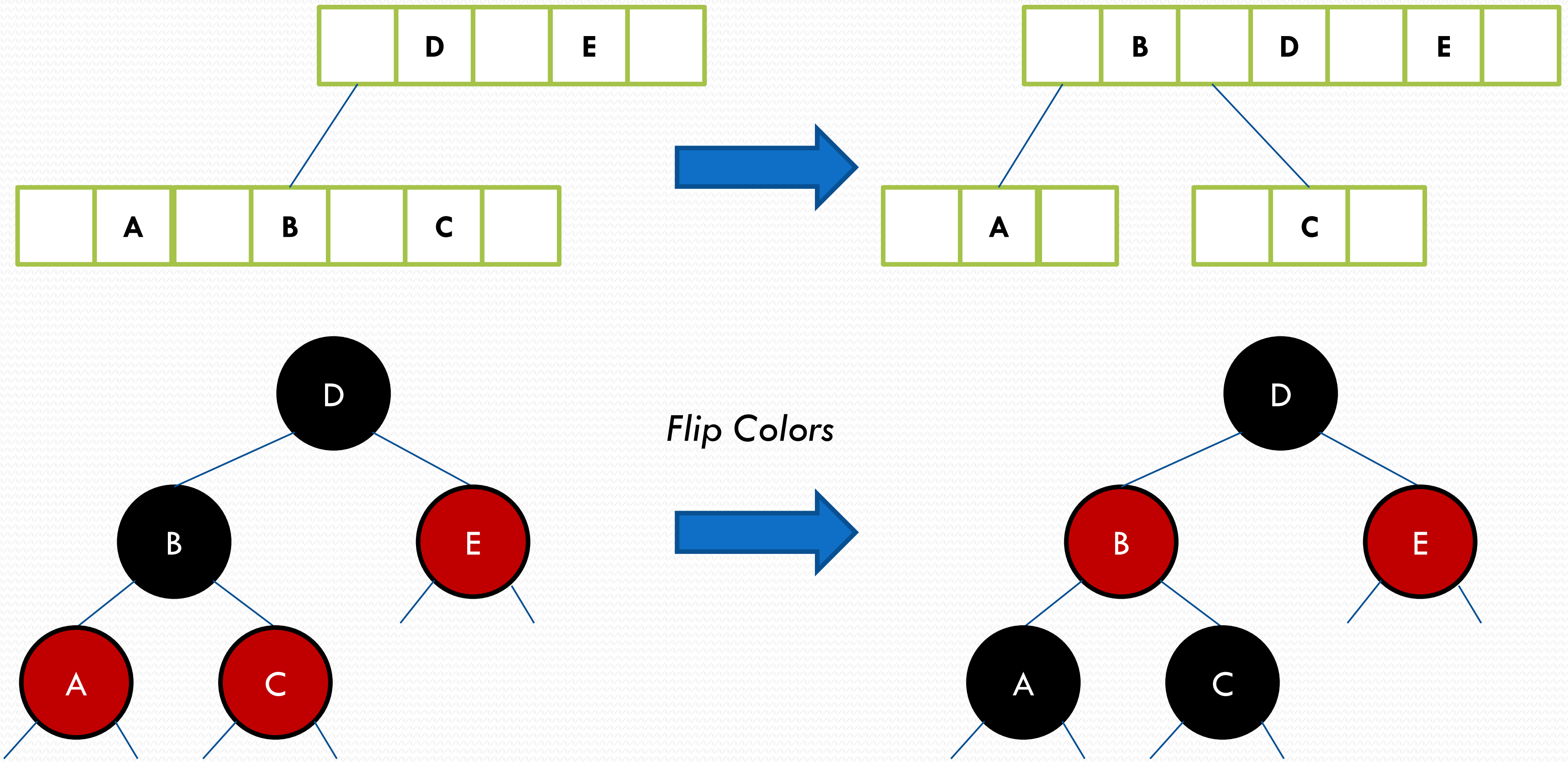




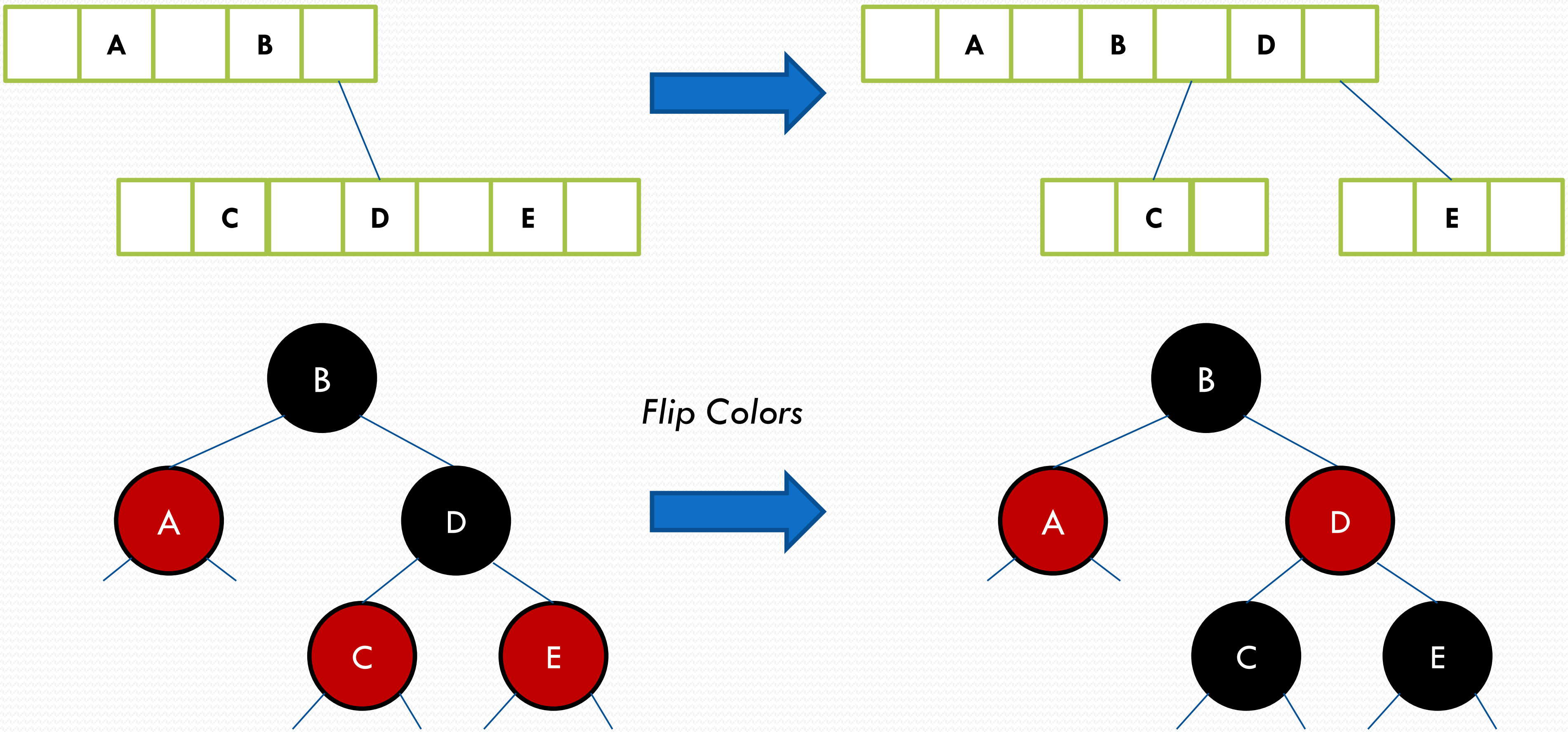
Splitting a 4-node connected to a 2-node (orientation #2):



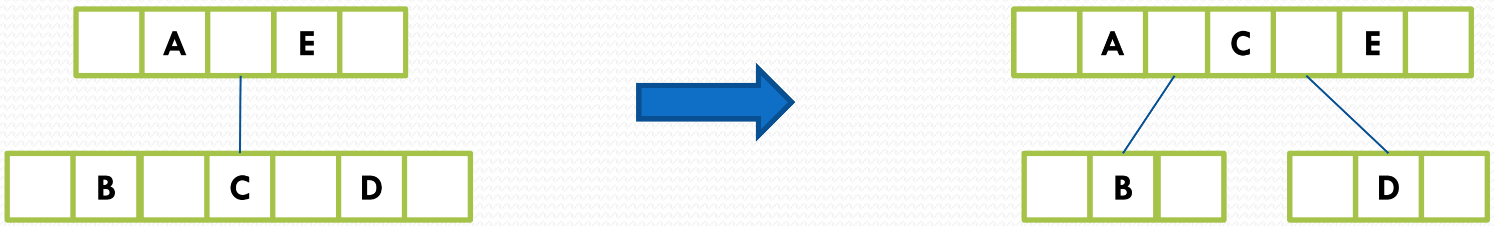
Splitting a 4-node connected to a 3-node (orientation #1):

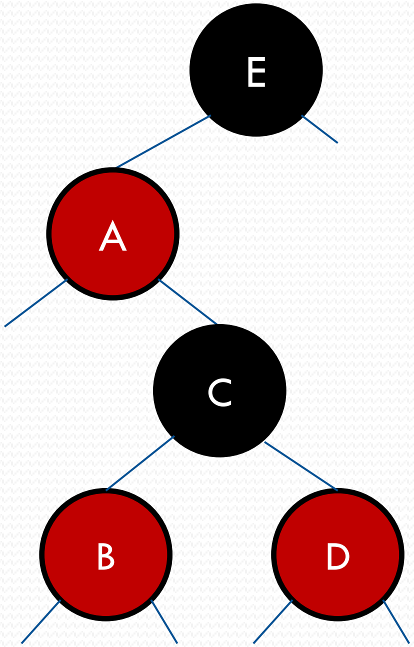


Splitting a 4-node connected to a 3-node (orientation #2):

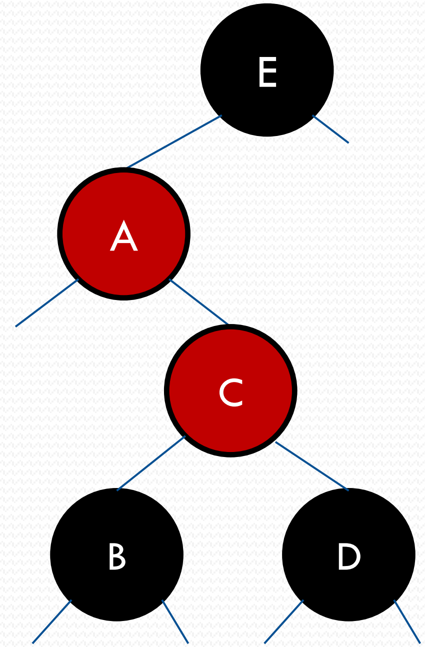


Splitting a 4-node connected to a 3-node (orientation #3):

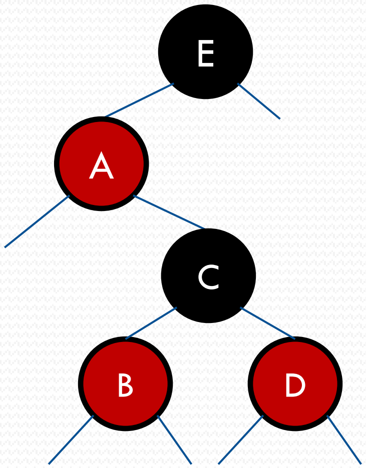




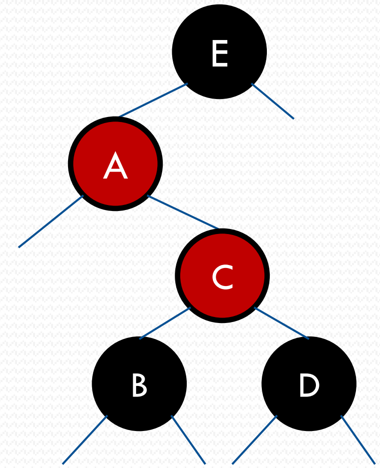
In this case, flipping colors is not enough



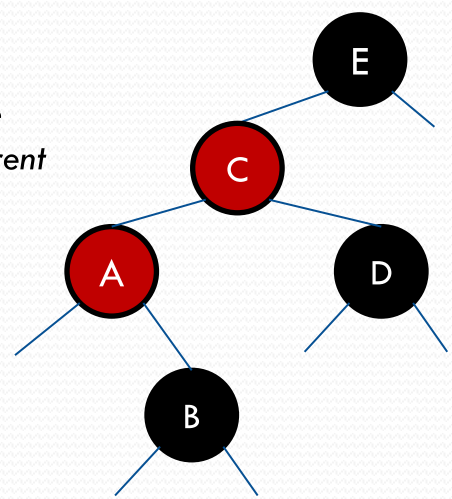
Splitting a 4-node connected to a 3-node (orientation #3):



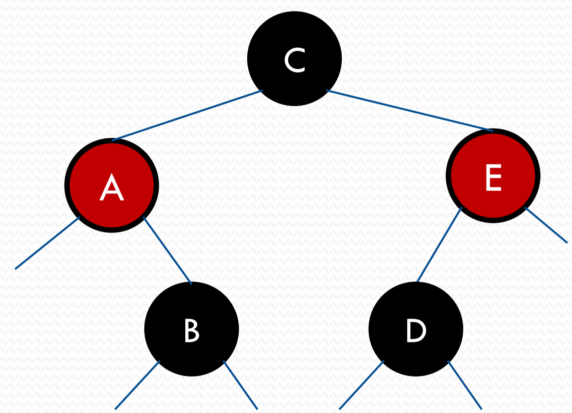
In this case, flipping colors is not enough



Promote node
Rotate about parent



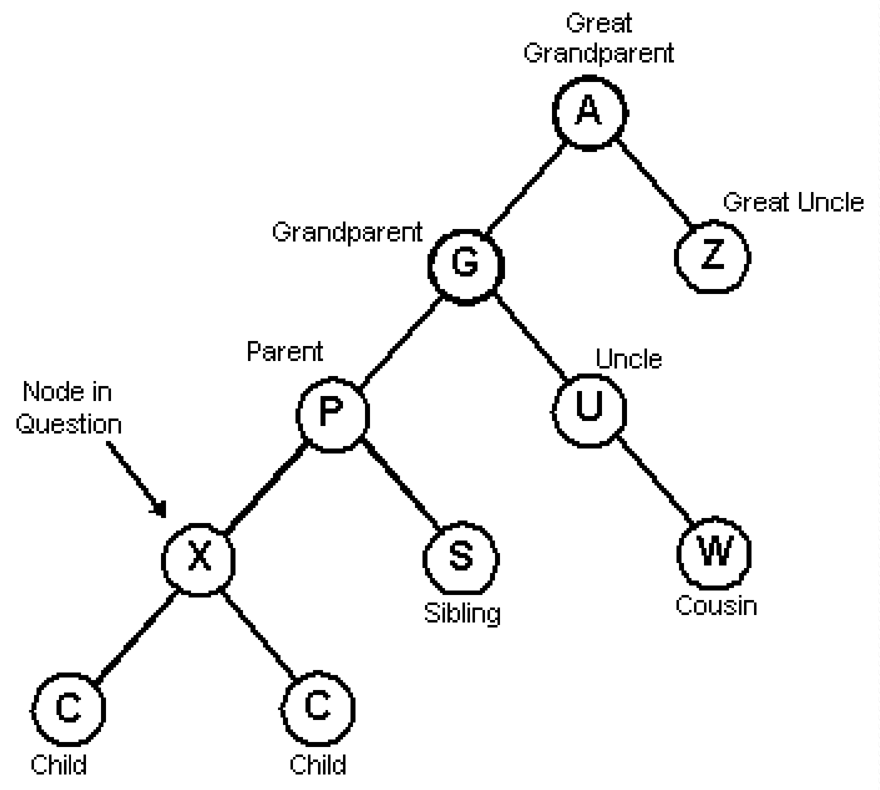
Promote node
Rotate about parent



Insertion

- Complexity with Red-Black Trees arises when an insertion destroys the Red-Black Tree properties:
  - Problem: Two **RED** nodes are adjacent.
    - This is because newly inserted nodes are always marked as **RED**, so if the parent is **RED** we have a "situation".

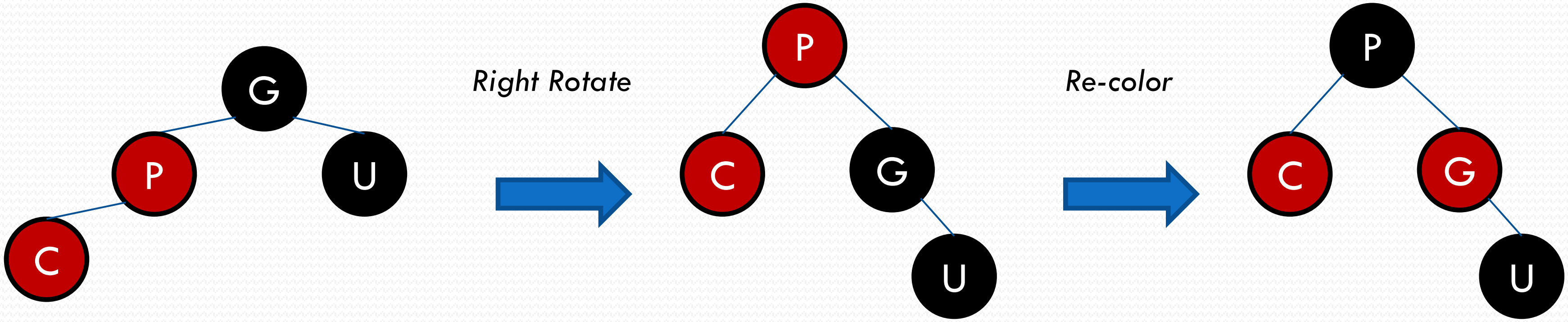
Terminology



Situation #1

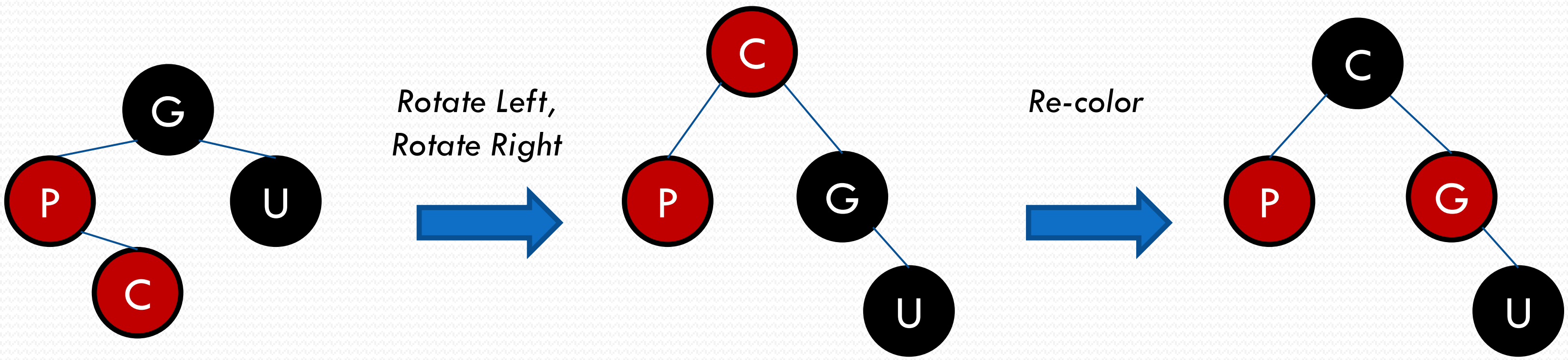
- Child and Parent are **RED** and Uncle is **BLACK**.
- Grandparent *must* be **BLACK** because tree was valid Red-Black before insertion
- **2 possible orientations** with the grandparent:

Orientation #1: (zig-zig)



- Rotate Grand-Parent (promote parent)
  - (G becomes child of P).
- Set Grand-Parent to **RED** and Parent to **BLACK**
- **Changes were local so we are done** (doesn't affect nodes above).

Orientation #2: (zig-zag)

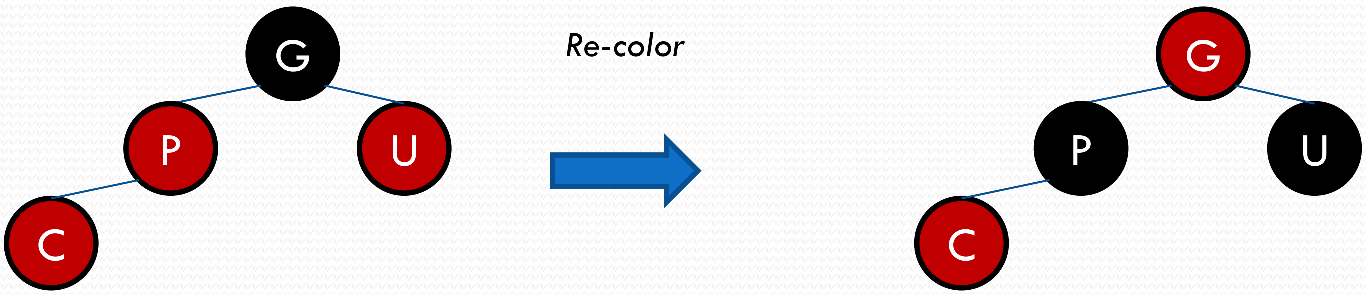


- Rotate Parent Left, then rotate Grand-Parent right(promote node, promote node).
- Set Grand-Parent to **RED** and Child to **BLACK.**
- **Changes were local so we are done.**

Situation #2

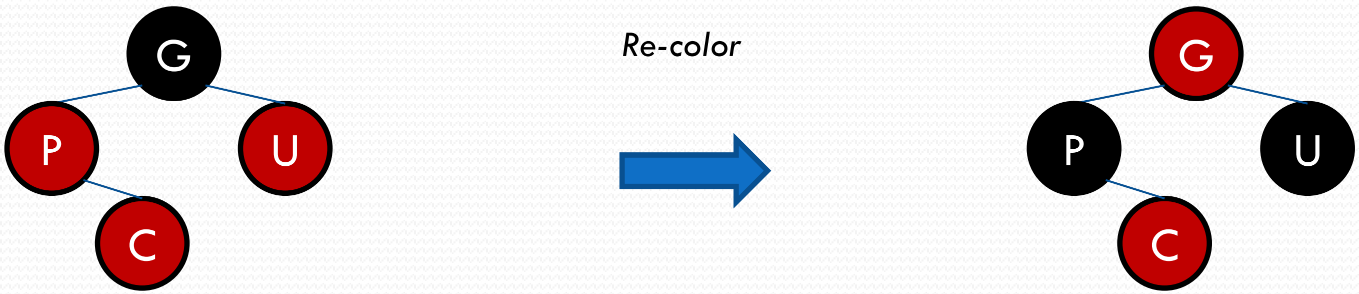
- Child and Parent are **RED** and Uncle is **RED**.
- Grandparent *must* be **BLACK** because tree was valid Red-Black before insertion
- **2 possible orientations** with the grandparent:

Orientation #1: (zig-zig)



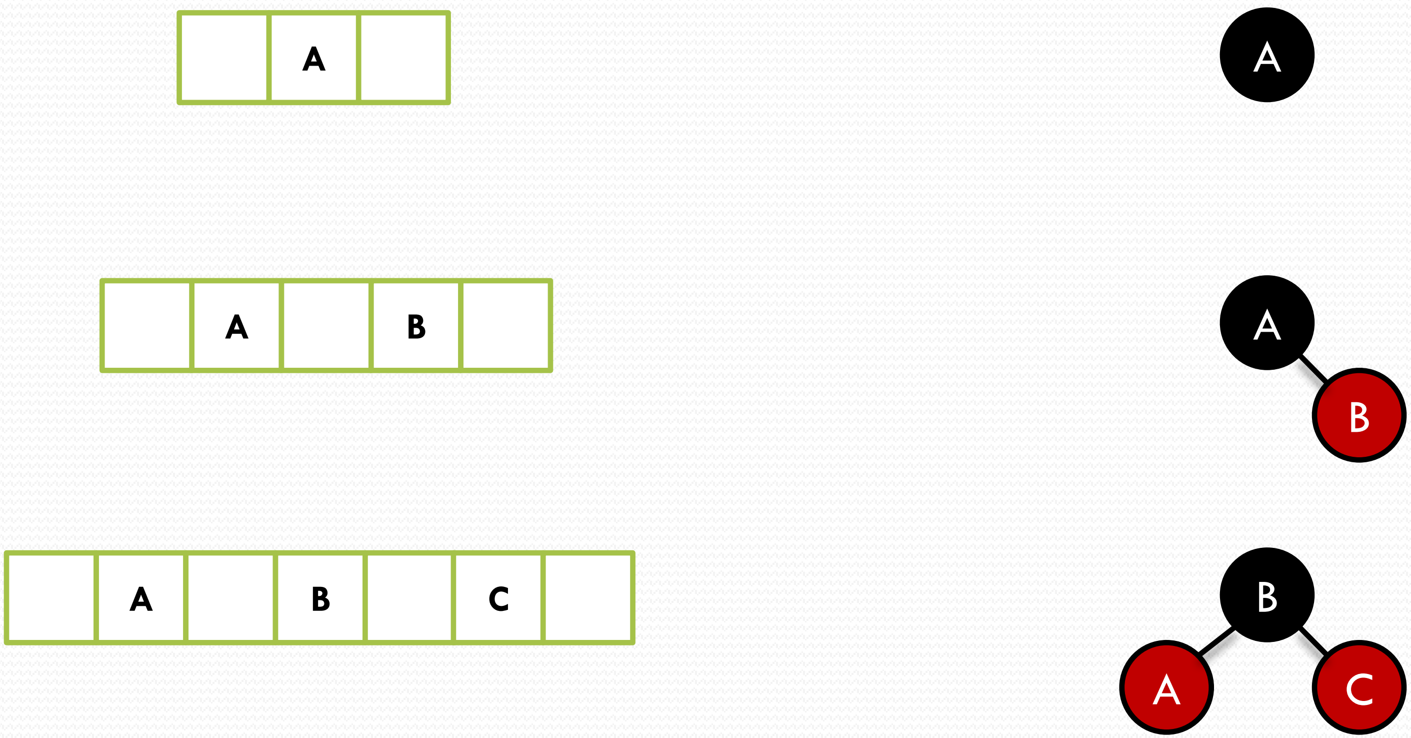
- Set Grand-Parent to **RED**, Parent and Uncle to **BLACK**
- **Changing G to RED** may affect **G**'s parent, so we need to continue up the tree.

Orientation #2: (zig-zag)

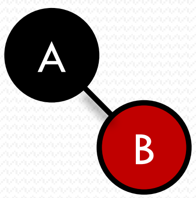


- Set Grand-Parent to **RED**, Parent and Uncle to **BLACK**
- **Changing G to RED** may affect **G**’s parent, so we need to continue up the tree.

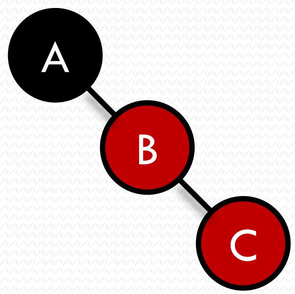
Insert in that order “A, B, C, D, E, F, G, H”



Inserting **C** in more details

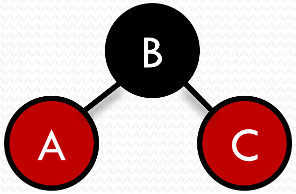


Original Tree

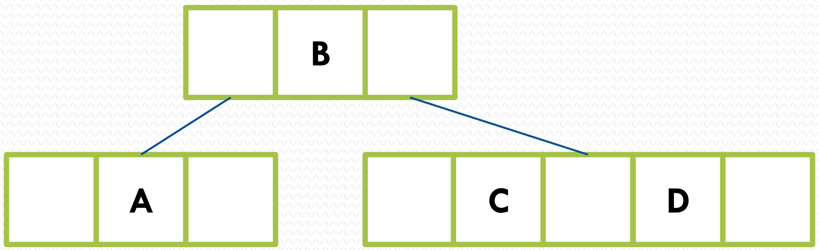


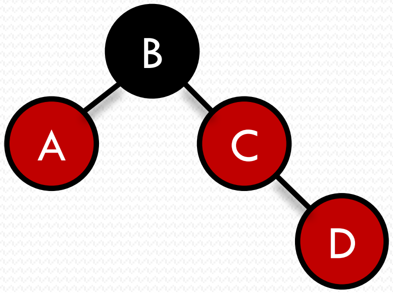
Situation 1.1
(parent red and uncle black)

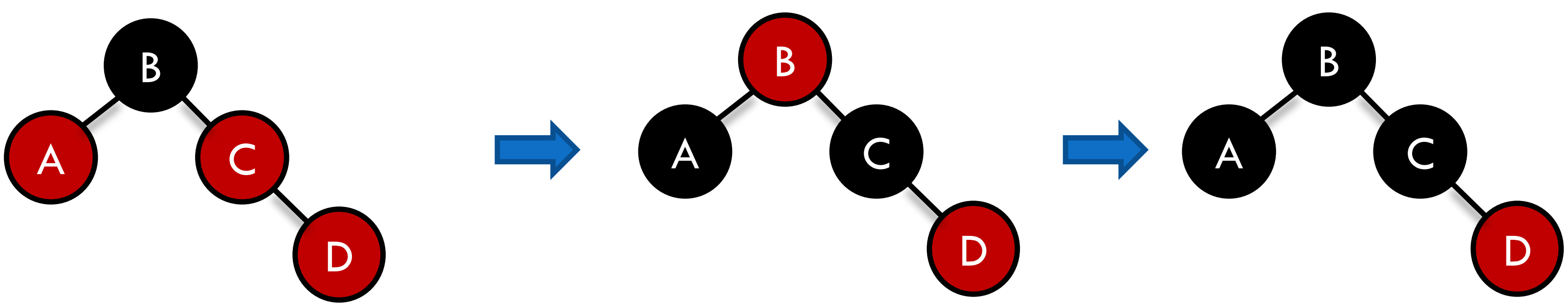
Promote parent
Set Grand-Parent (A) to Red
and parent (B) black



Insert in that order “A, B, C, D, E, F, G, H”





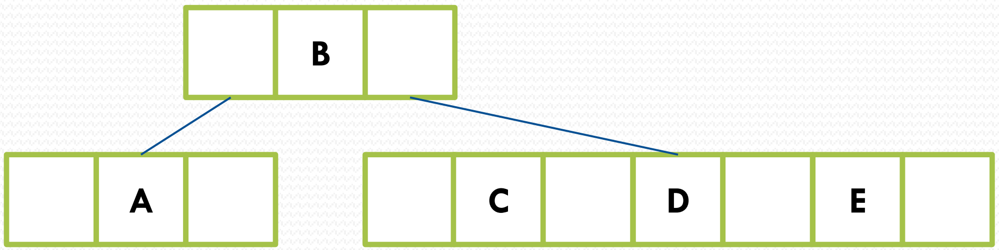


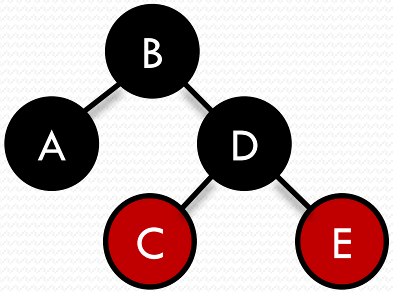
Situation 2.1
(parent red and uncle red)

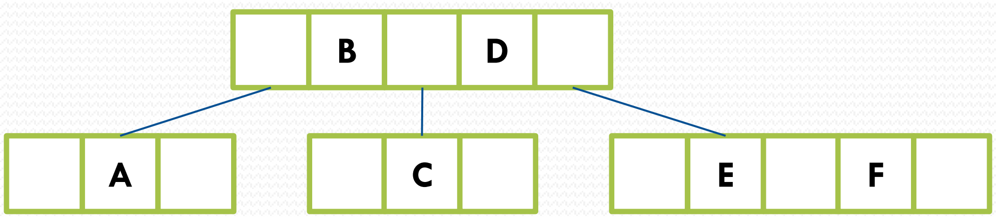
Flip Colors

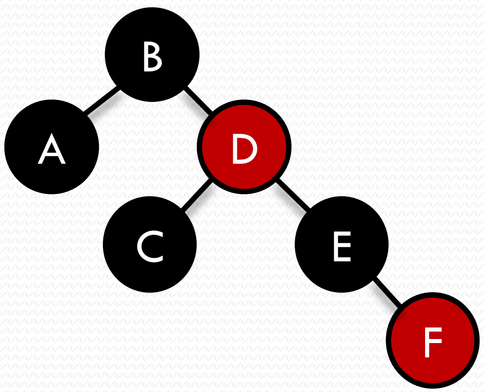
Root is always black
(our convention)

Insert in that order “A, B, C, D, E, F, G, H”

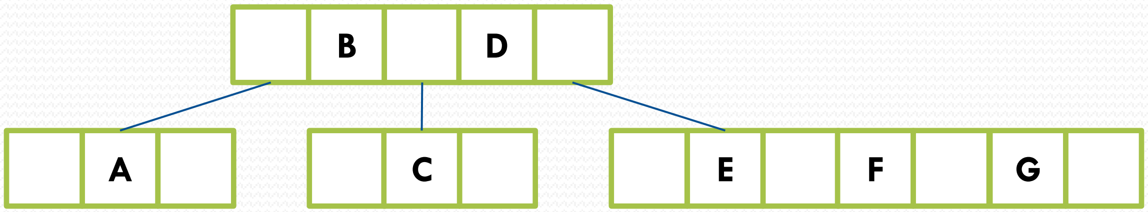


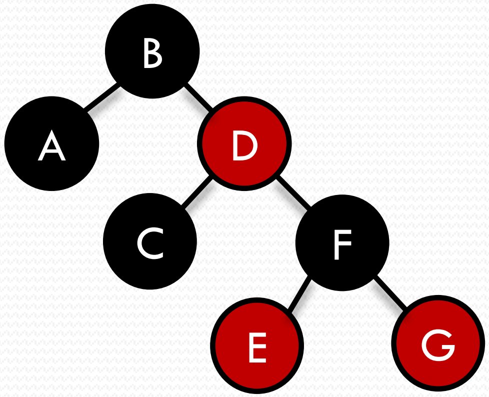




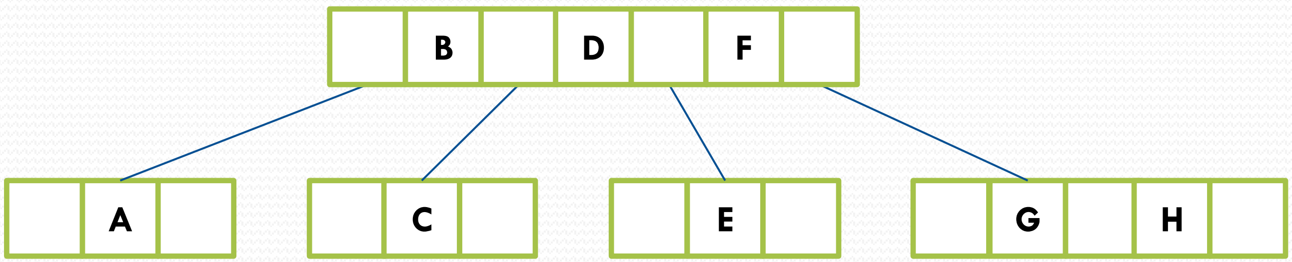


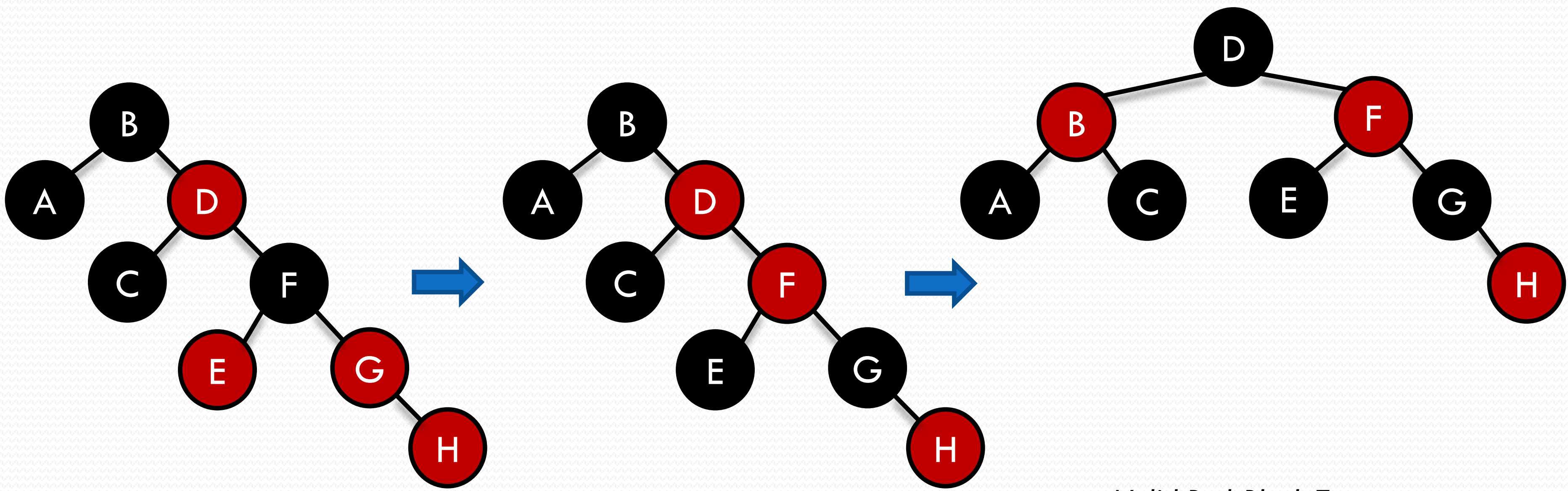
Insert in that order “A, B, C, D, E, F, G, H”





Insert in that order "A, B, C, D, E, F, G, H"





Situation 2.1

Situation 1.1

Valid Red-Black Tree

Summary

- Red-Black trees are BSTs, so standard BST search algorithms work as-is.
- They correspond directly to 2-3-4 trees, so they remain (approximately) balanced after inserting.
- There is less work during searches because no balancing is done (split on the way down).
  - We only balance if we add a node.
- The insertion/rebalancing algorithm is fairly simple.
- Searching, inserting, and re-balancing are all **O(lg N)**.

Summary (2)

- Red-Black trees ensure the underlying 2-3-4 tree is balanced
  1. The corresponding 2-3-4 tree is exactly balanced and requires at most lg N comparisons to reach a leaf. The worst case complexity, then, is O(lg N).
  2. The Red-Black tree is approximately balanced and requires at most 2 lg N comparisons to reach a leaf. The worst case complexity, then, is O(lg N). On average, the number of comparisons is 1.002 lg N.