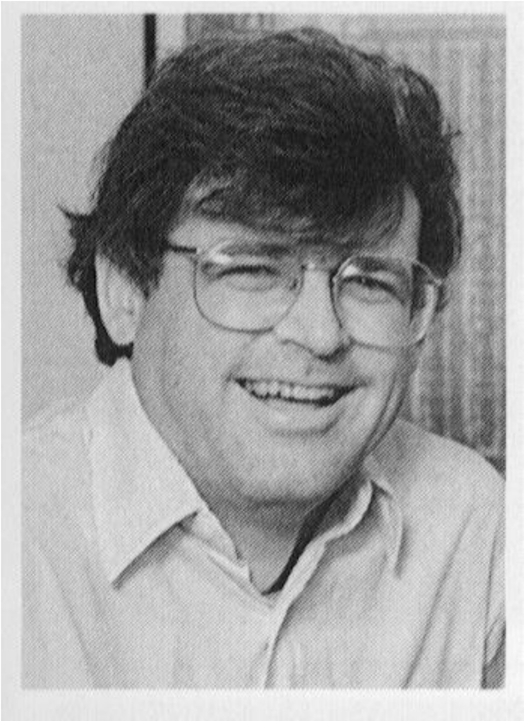
Red-Black Trees

Leonidas J. Guibas
Professor,
Stanford University



Robert Sedgewick
Professor
Princeton University

Leonidas J. Guibas and Robert Sedgewick (1978). "A Dichromatic Framework for Balanced Trees". Proceedings of the 19th Annual Symposium on Foundations of Computer Science. pp. 8–21. doi:10.1109/SFCS.1978.3

Red-Black Trees

- 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(log N).
- The insertion/re-balancing algorithm is fairly simple. However, coming up with the algorithm is not.

Properties of Red-Black Trees 1

- 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.

Properties of Red-Black Trees

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

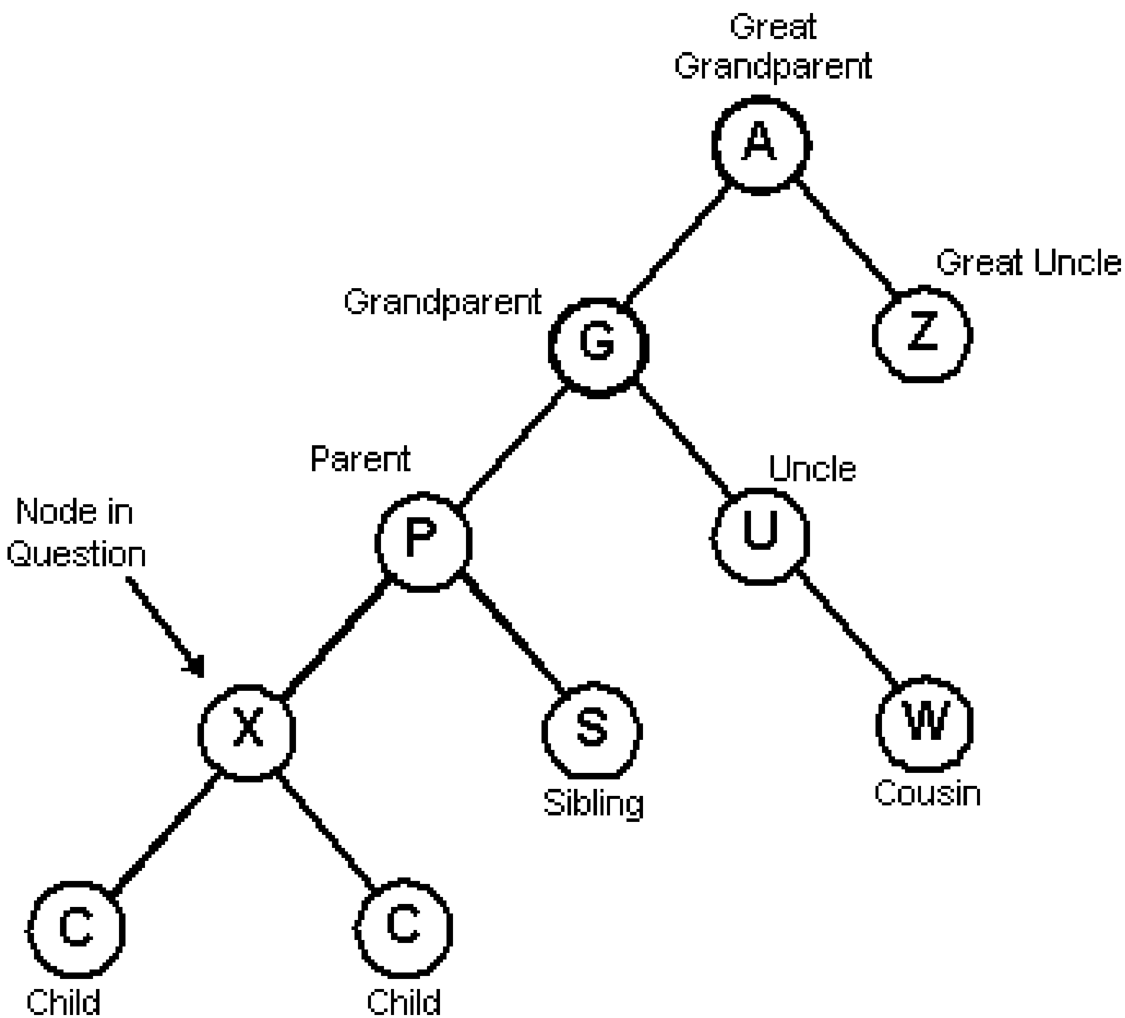
Properties of Red-Black Trees 2

- Each node is marked as **RED** or **BLACK.**
- All leaves and **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 node 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.

Terminology



Insertion

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”.

Insertion (Parent is RED)

- Uncle is **RED**
- GP will be **BLACK**

1. Recolor
   - Set GP to **RED**
   - P to **BLACK**
   - U to **BLACK**
2. If GP is **RED.** It might disturb check with the color of GGP

- Uncle is **BLACK or NULL**
- GP will be **BLACK**

OR1 (ZIG-ZIG)

1. Promote P
2. Recolor
   - Set GP to **RED**
   - P to **BLACK**

OR2 (ZIG-ZAG)

1. Promote C
   Promote C
2. Recolor
   - Set GP to **RED**
   - C to **BLACK**

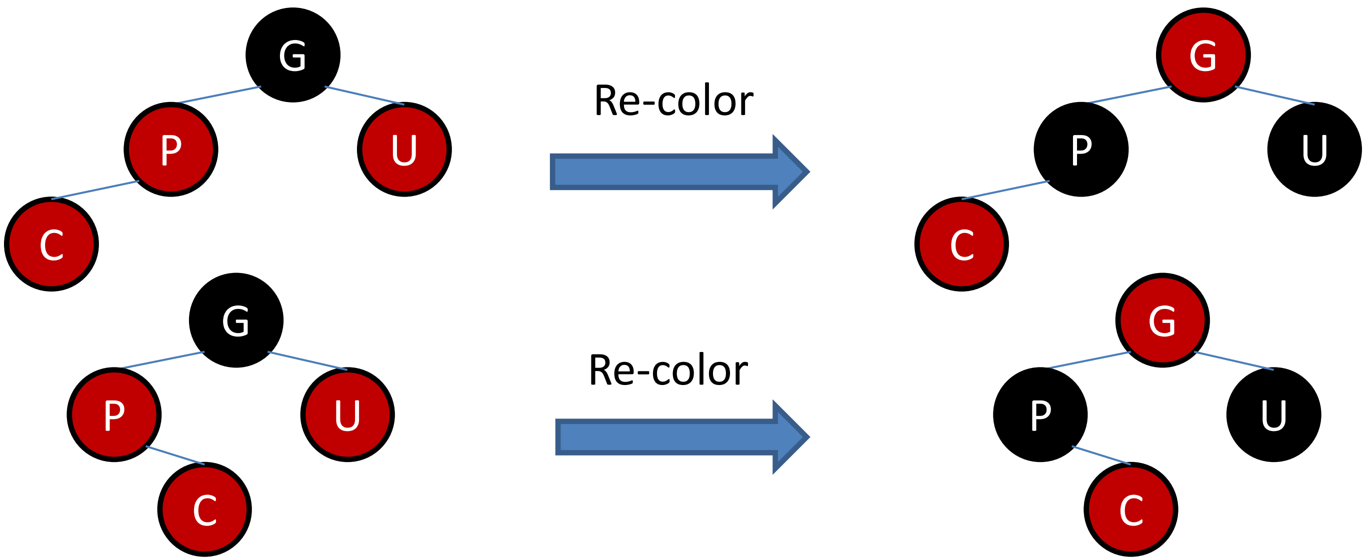
In this case changes are local, so no need for any further changes in the tree

Insertion: Situation 1

- Child and Parent are **RED** and Uncle is **RED**.

- Grandparent must be **BLACK** because tree was a valid Red-Black before insertion.

Insertion: Situation 1

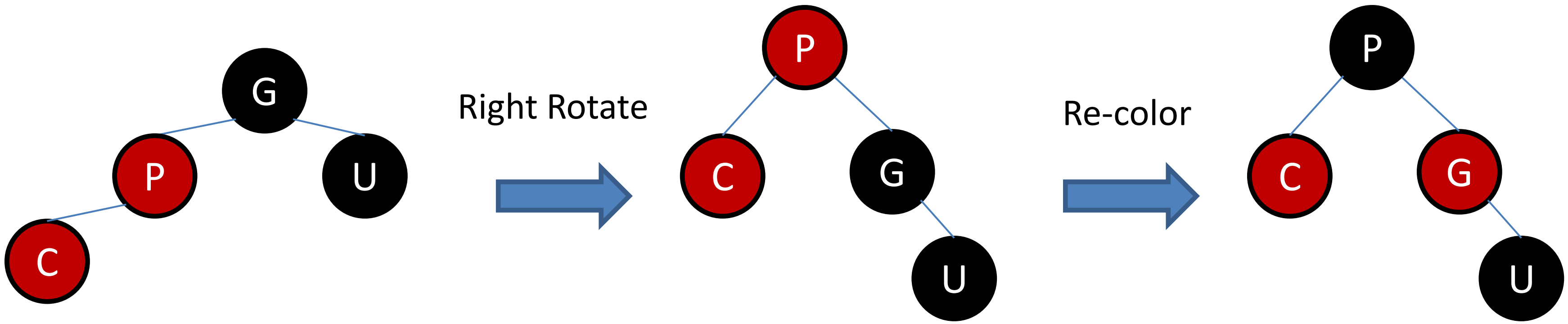


- Set Grand-Parent to **RED**, Parent and Uncle to **BLACK**
- **Changing G to RED** may affect **G**'s parent, so we need to continue up the tree.

Insertion: Situation 2

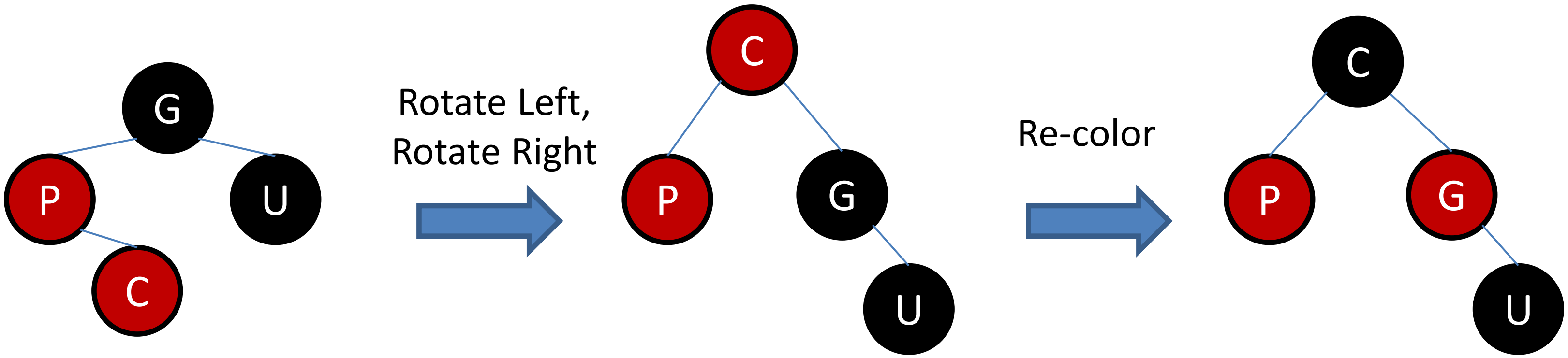
- Child and Parent are **RED** and Uncle is **BLACK**.
- Grandparent must be **BLACK** because tree was valid Red-Black before insertion
- **2 possible orientations** with the grandparent

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).

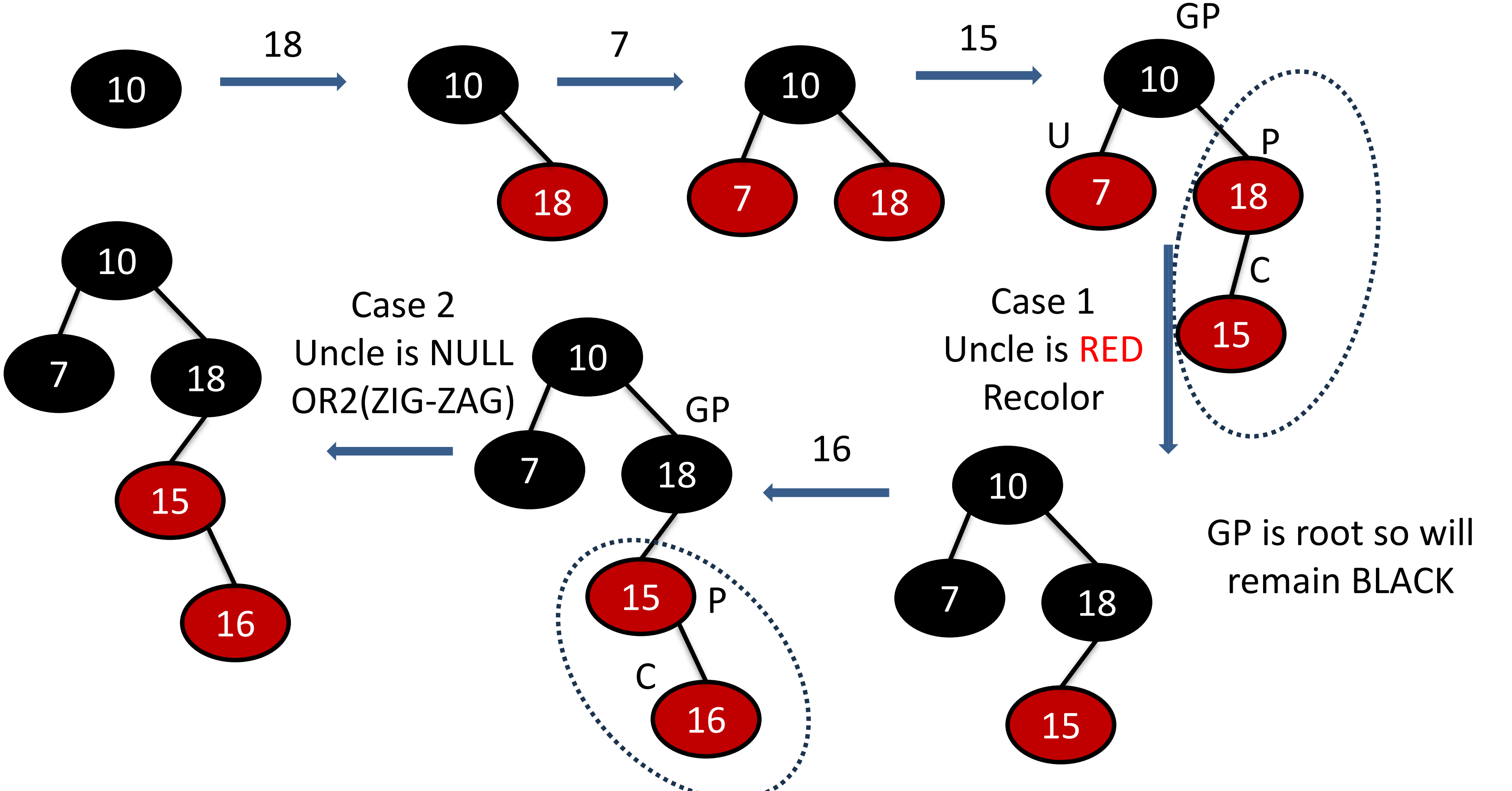
Orientation #2: (Zig-Zag)



- 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.**

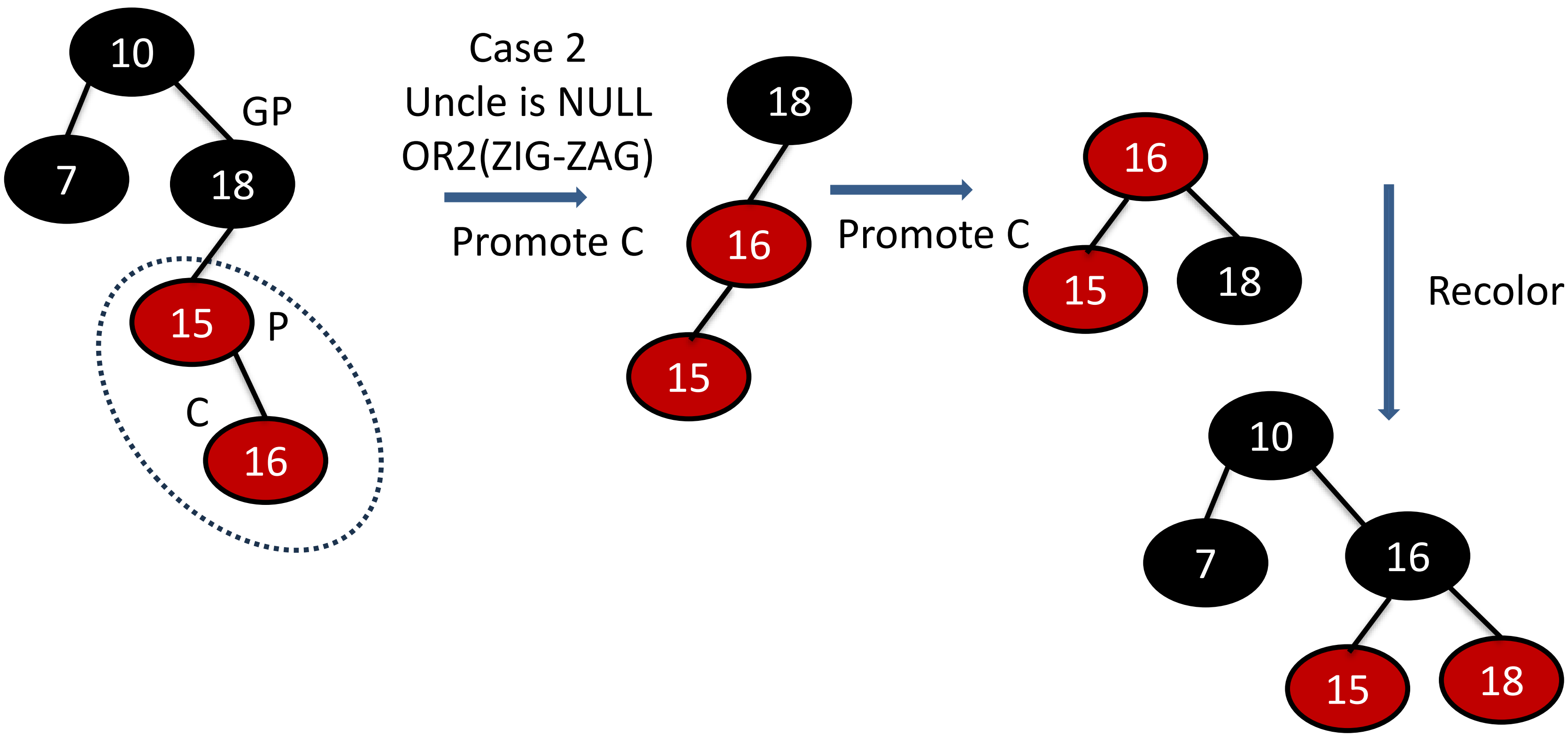
Insertion

Insert 10, 18, 7, 15, 16, 30, 25, 40



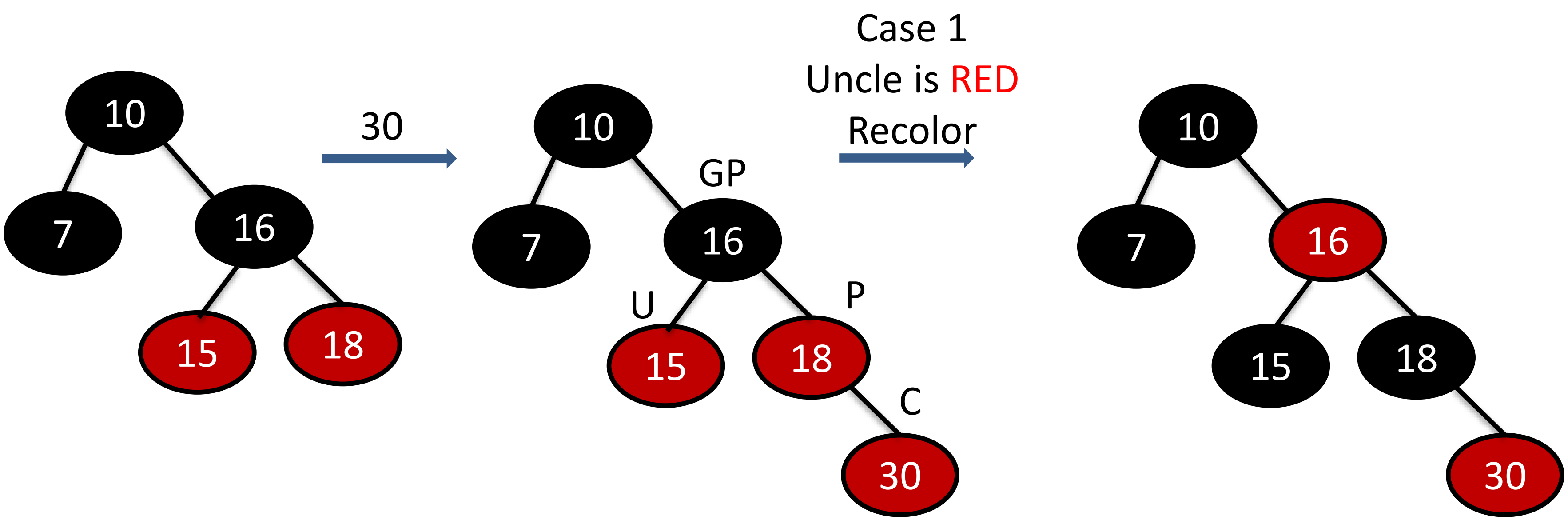
Insertion

Insert 10, 18, 7, 15, 16, 30, 25, 40



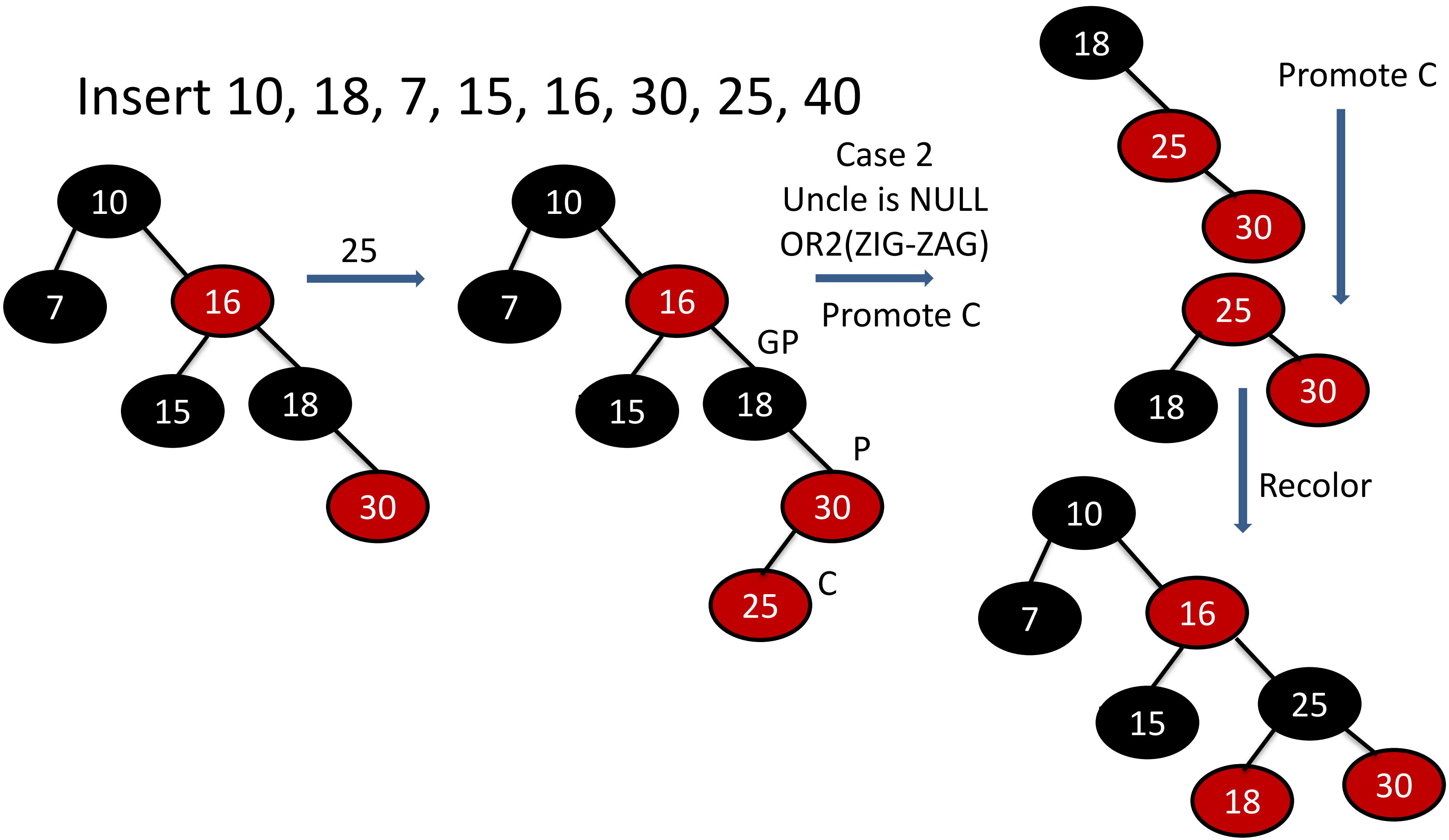
Insertion

Insert 10, 18, 7, 15, 16, 30, 25, 40



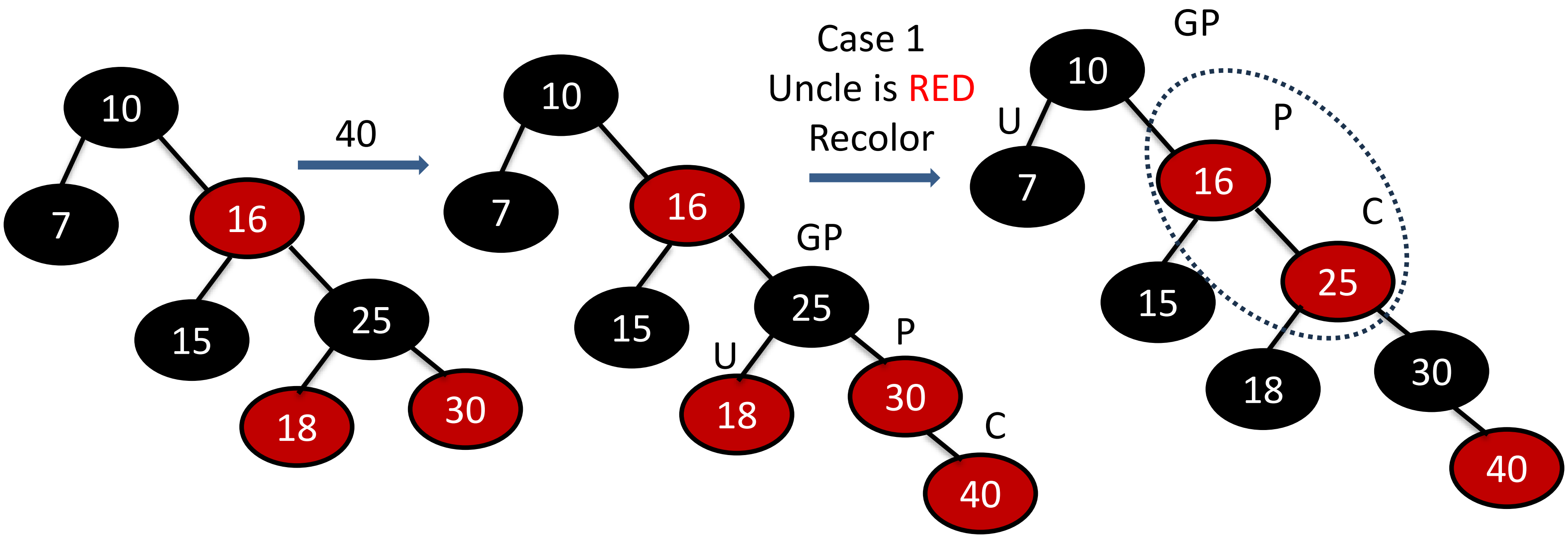
Insertion

Insert 10, 18, 7, 15, 16, 30, 25, 40



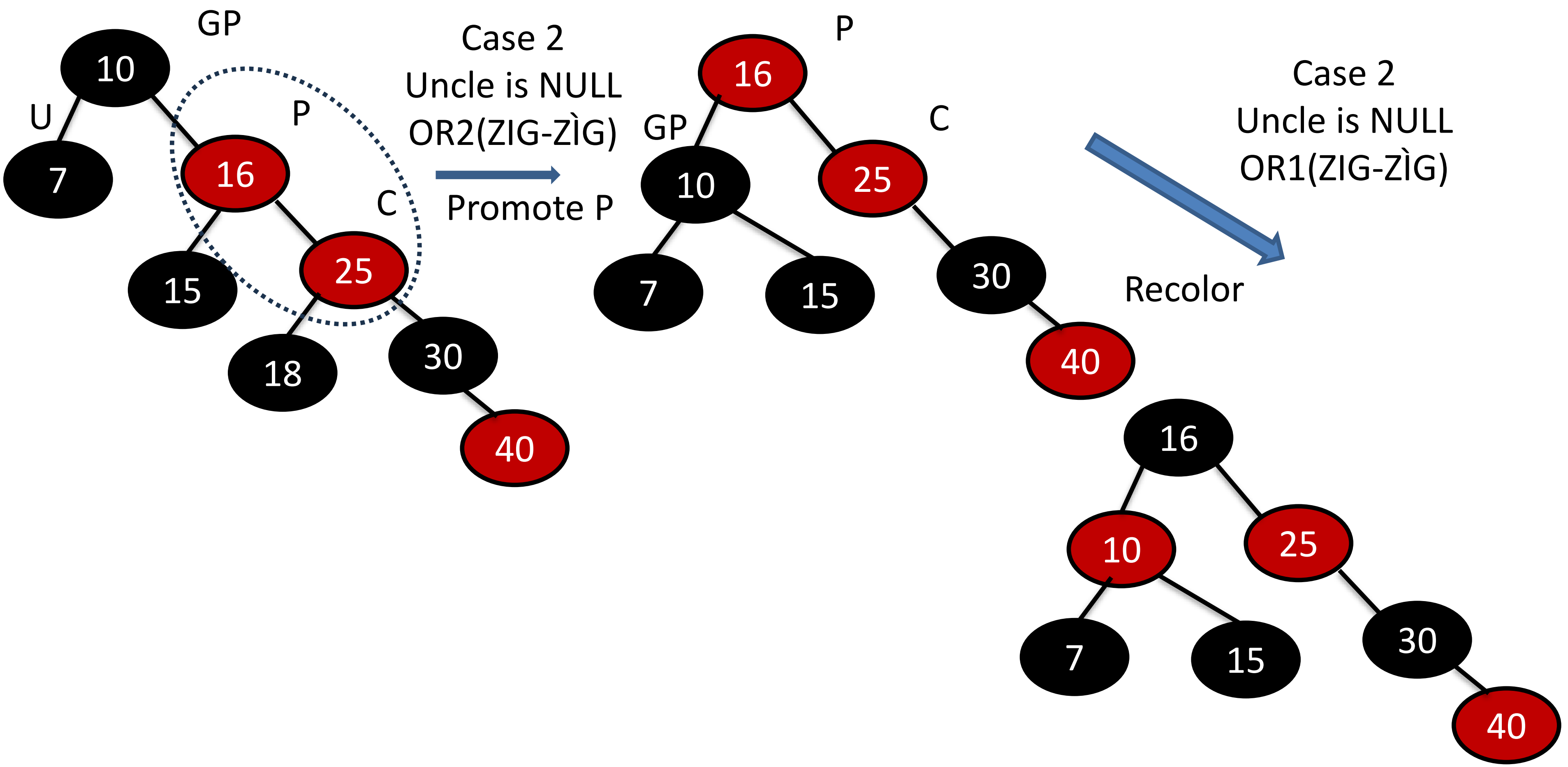
Insertion

Insert 10, 18, 7, 15, 16, 30, 25, 40



Insertion

Insert 10, 18, 7, 15, 16, 30, 25, 40



Mapping 2-3-4 Trees into R-B Trees

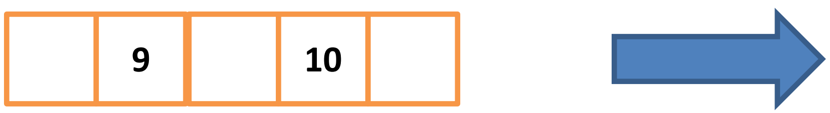
- Red-Black Trees are used to represent 2-3-4 trees in a BST form.

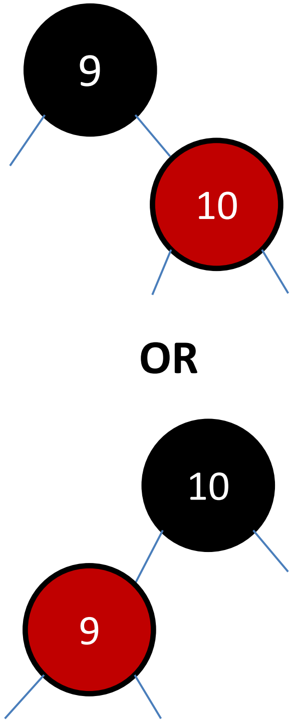
- It is possible to map any 2-3-4 Trees into a Red-Black Tree and vice versa.

- There are several situations
  - 2 – node
  - 3 – node
  - 4 – node
  - 2-node connected to 3-nodes
  - 3-node connected to 4-nodes

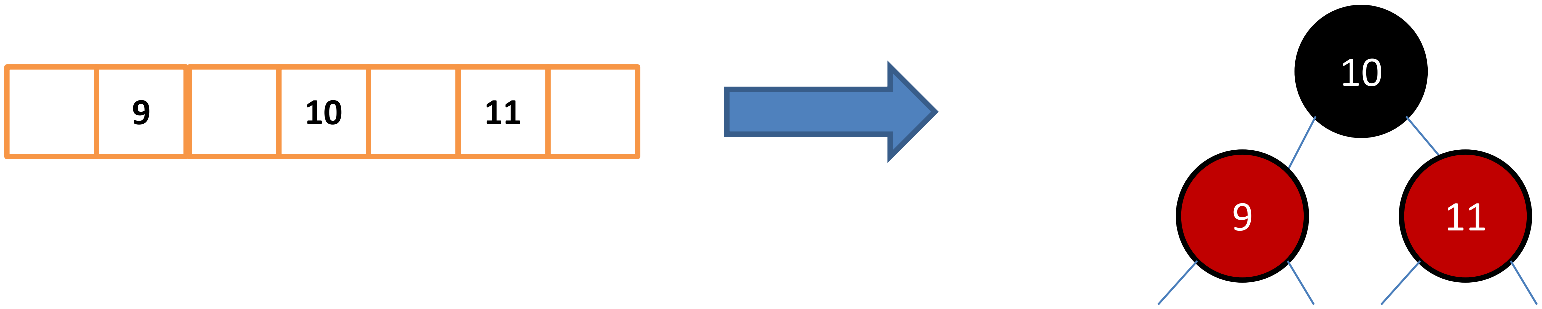
2-Node to R-B Tree Node

3-Node to R-B Tree Node

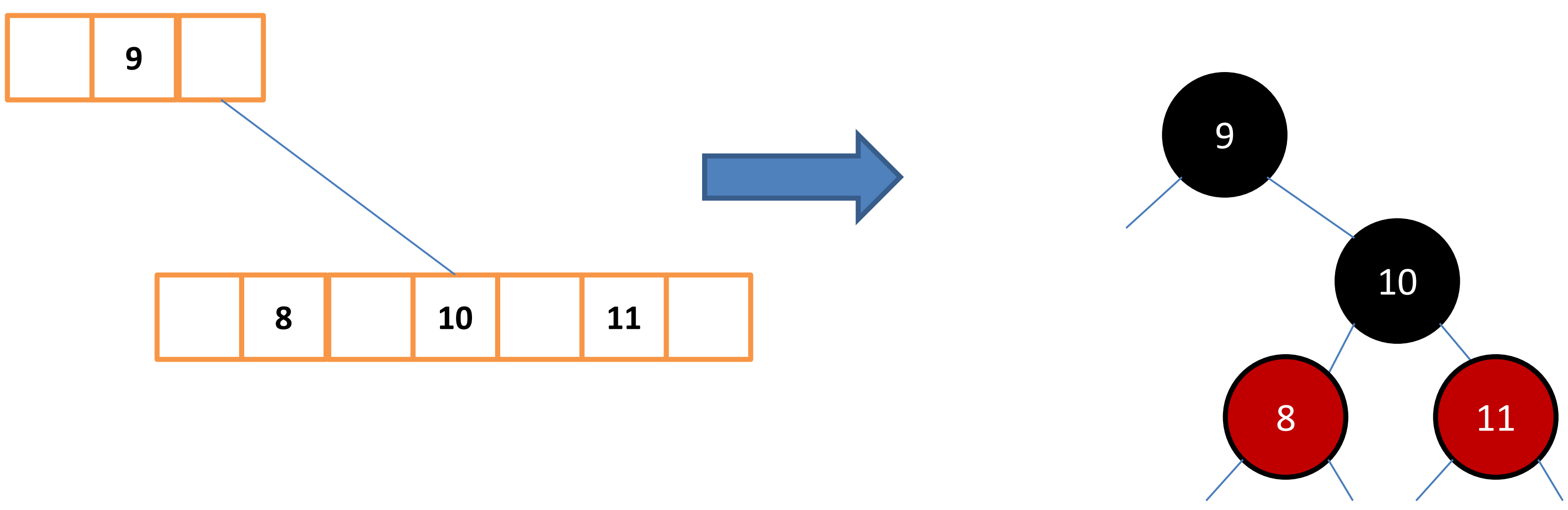




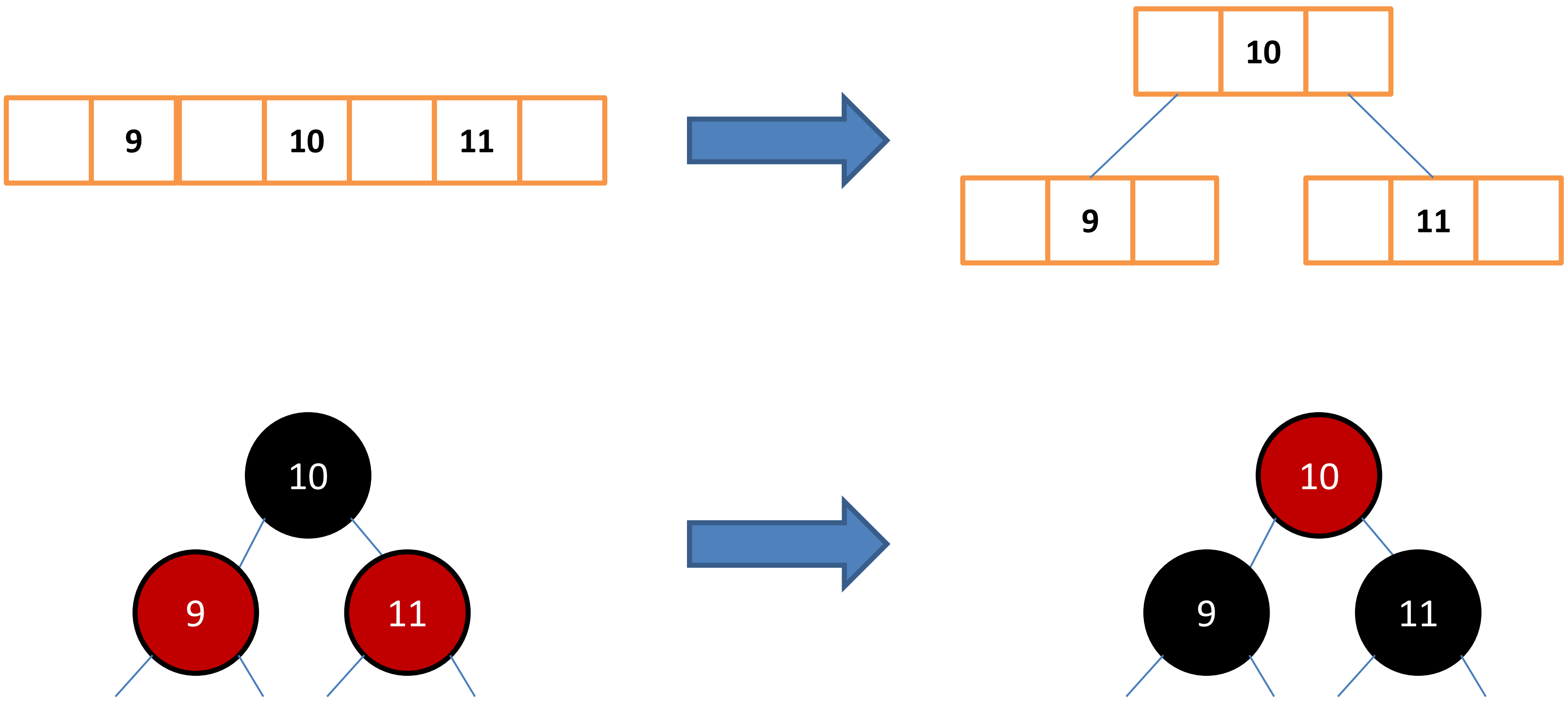
4-Node to R-B Tree Node



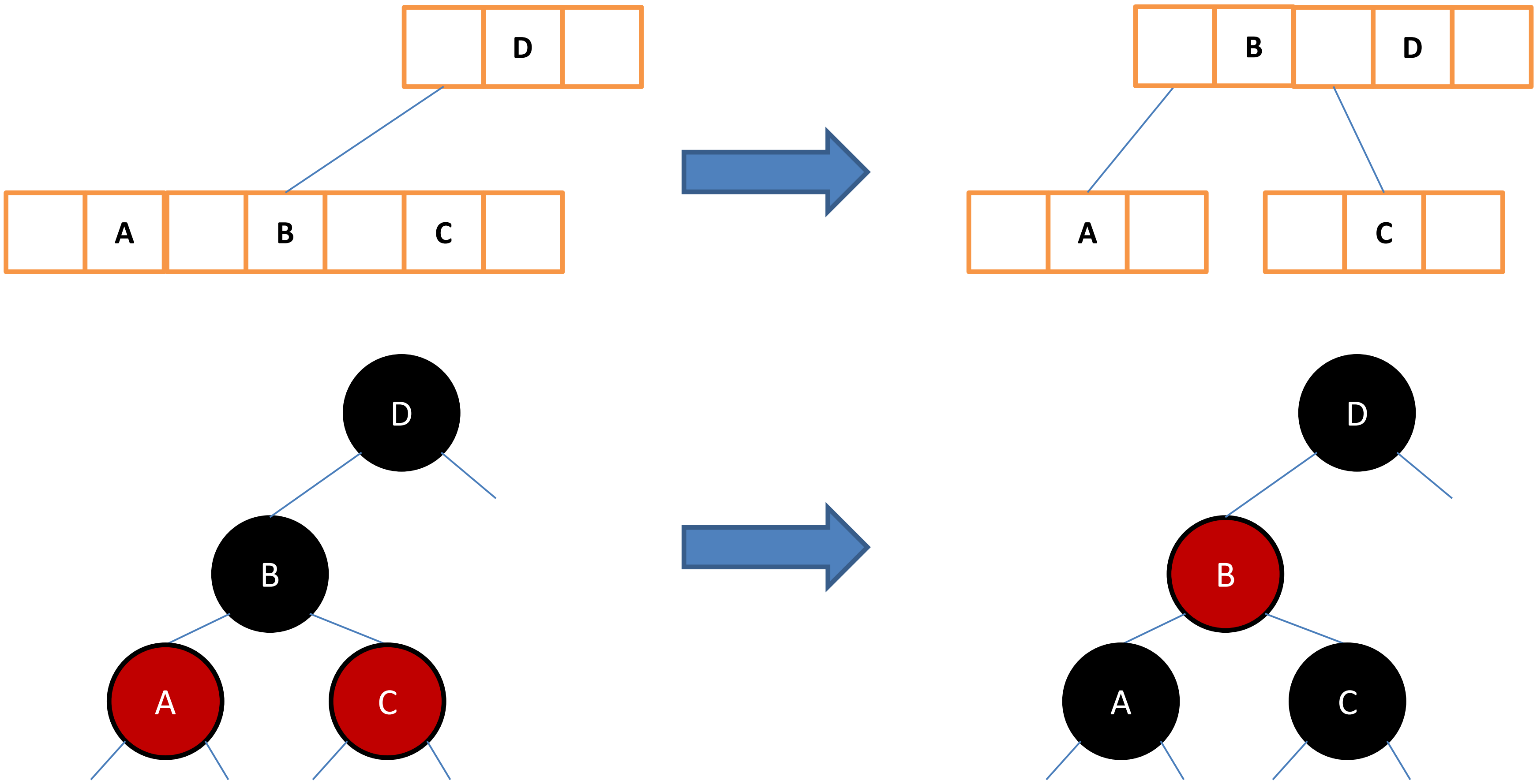
2-Node Connected to a 4-Node



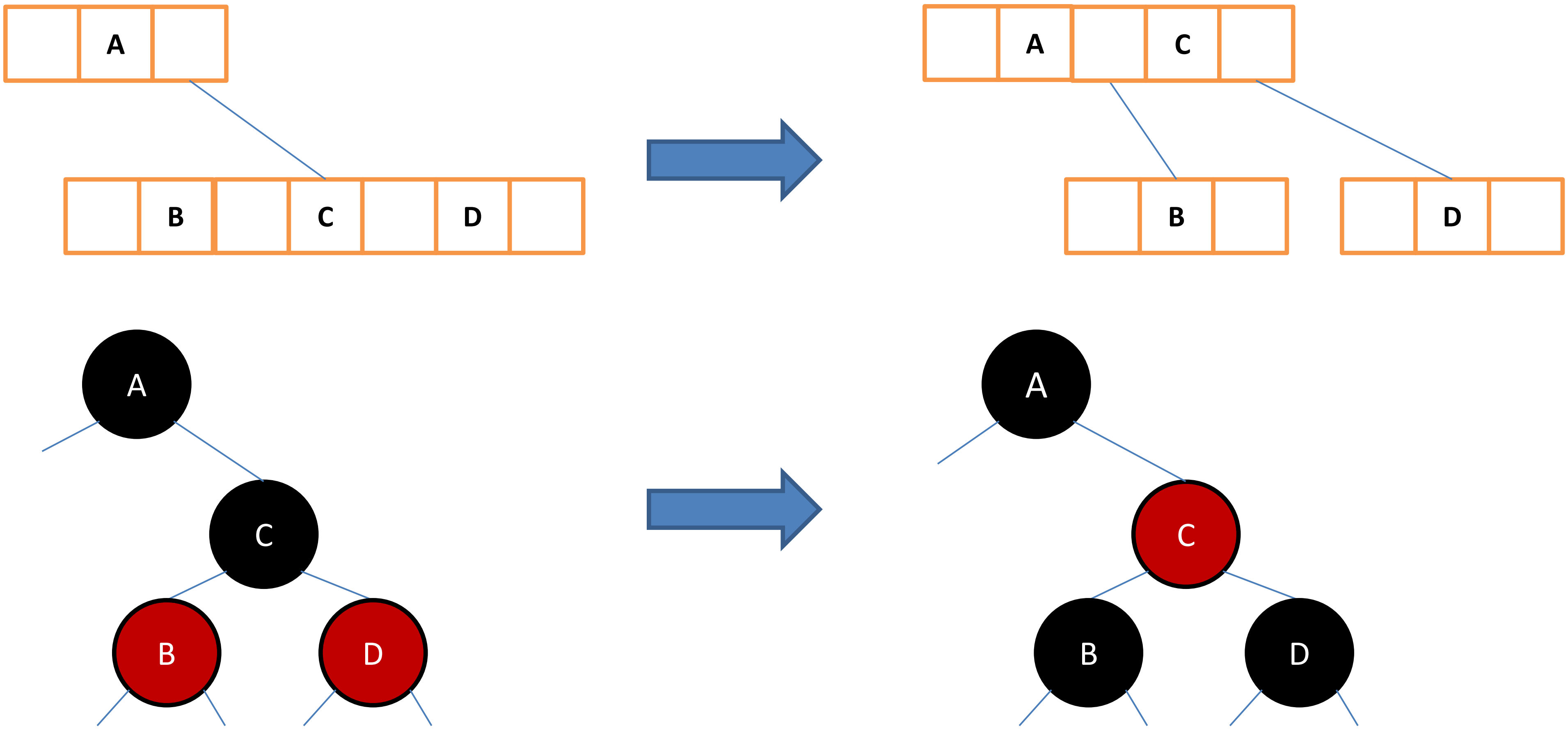
Splitting a 4-node:



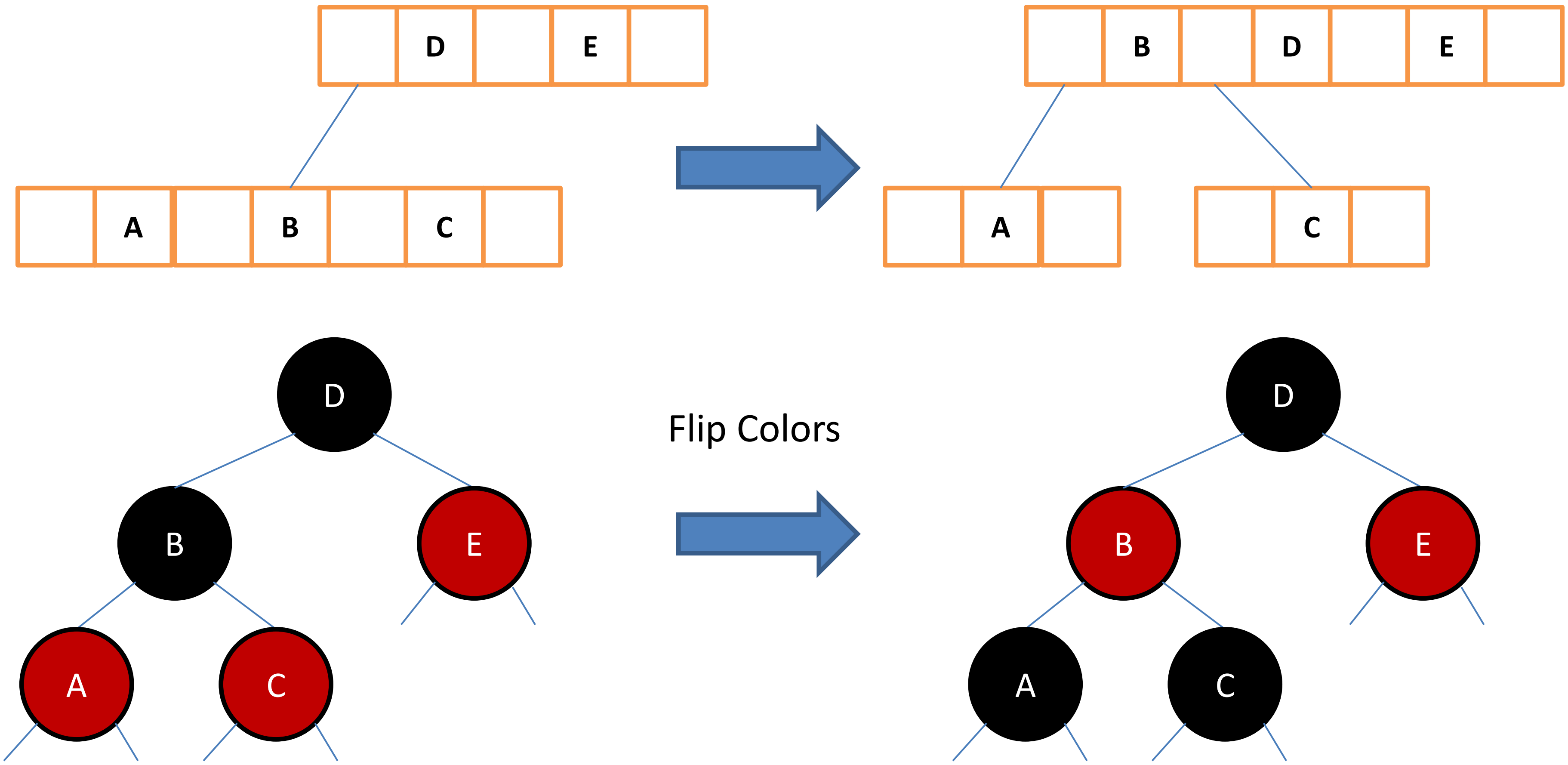
Splitting a 4-node connected to a 2-node (orientation #1):



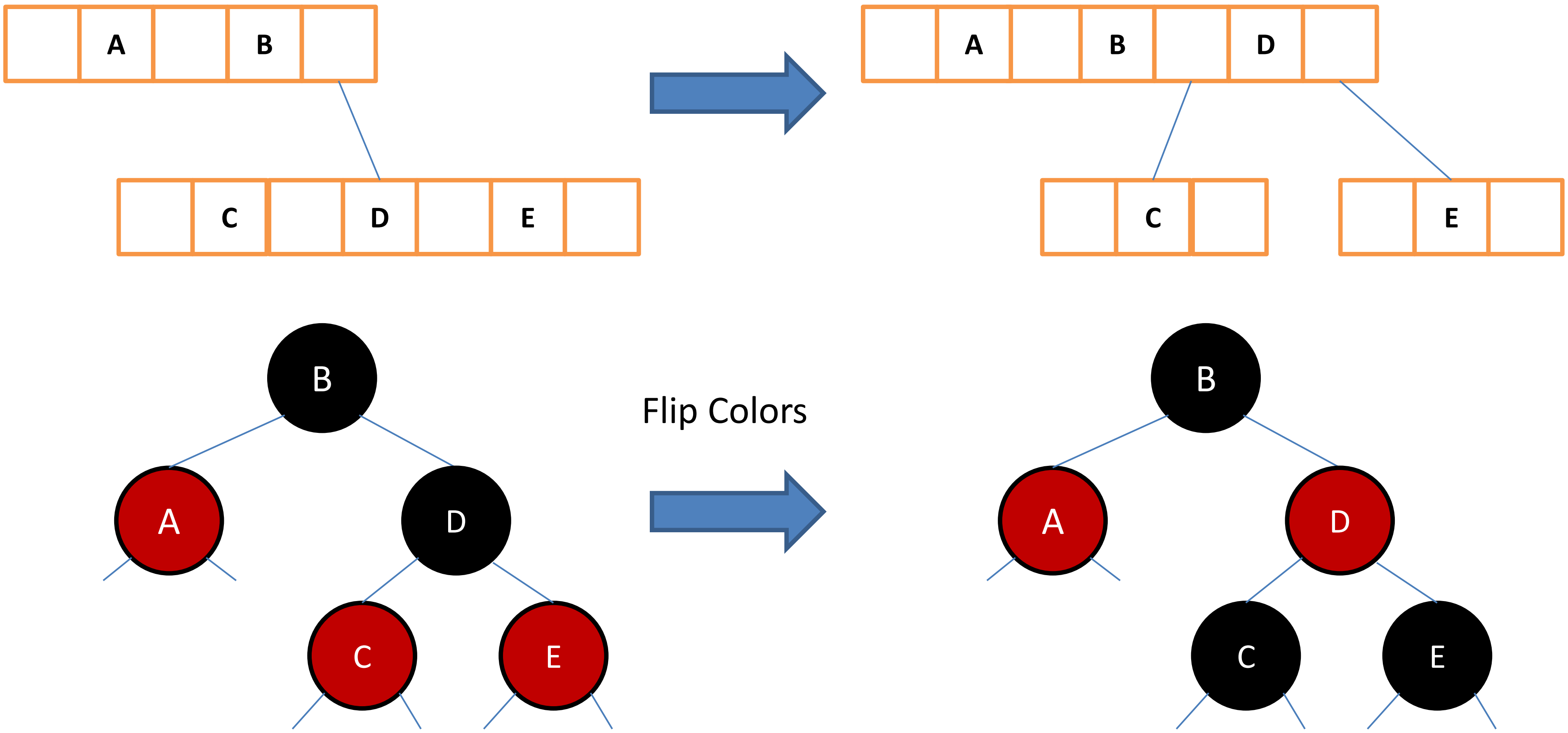
Splitting a 4-node connected to a 2-node (orientation #2):



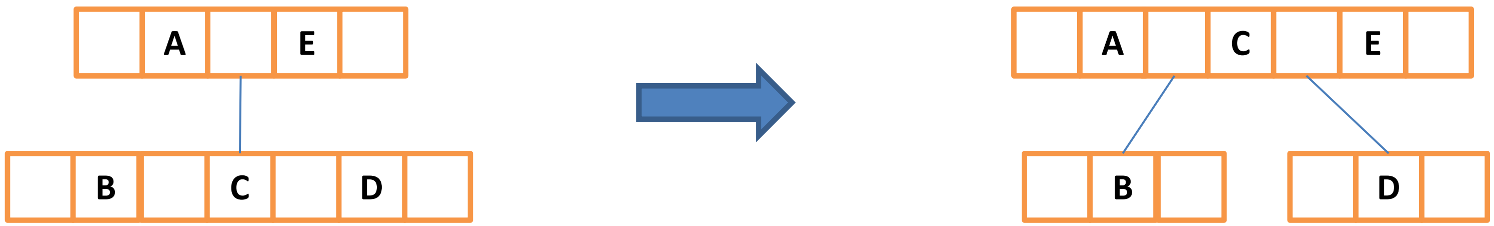
Splitting a 4-node connected to a 3-node (orientation #1):

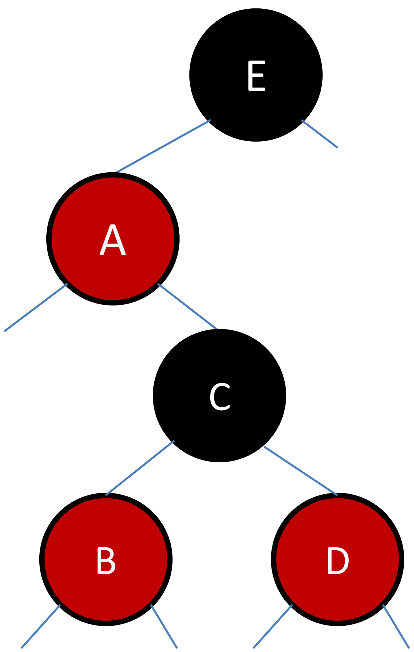


Splitting a 4-node connected to a 3-node (orientation #2):

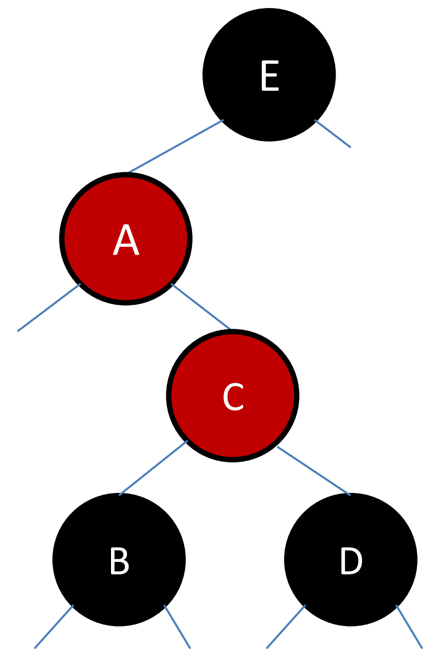


Splitting a 4-node connected to a 3-node (orientation #3):

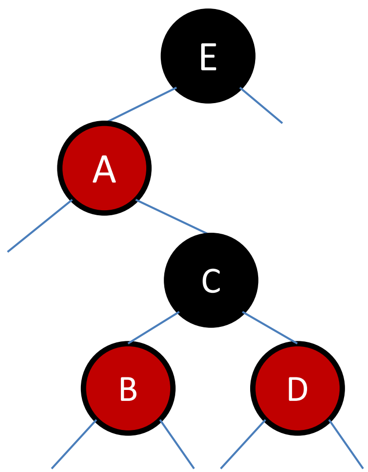




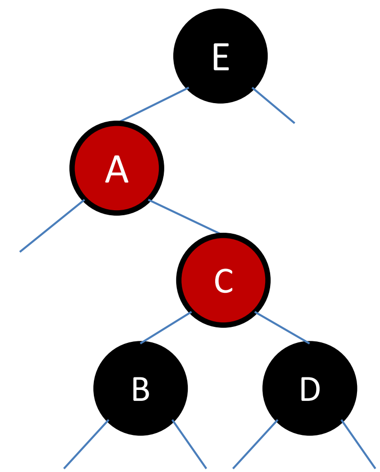
In this case, flipping colors is not enough



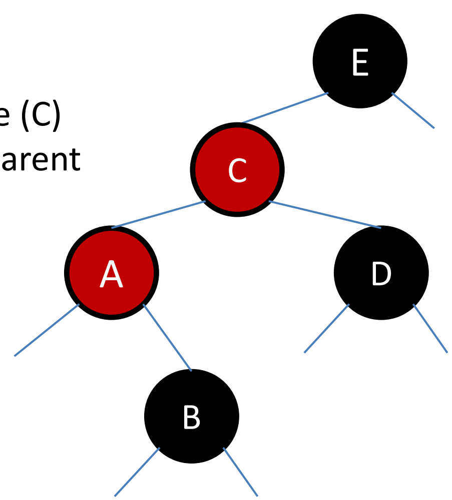
Splitting a 4-node connected to a 3-node (orientation #3):



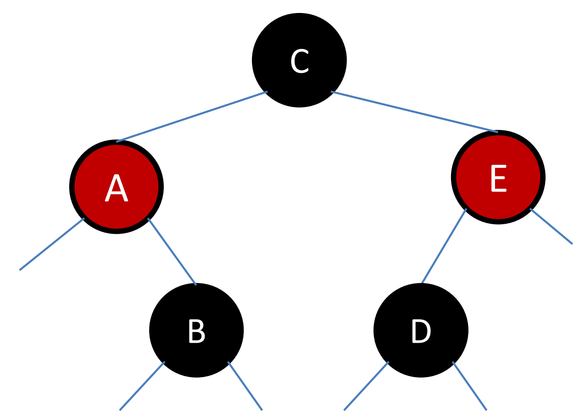
In this case, flipping colors
is not enough



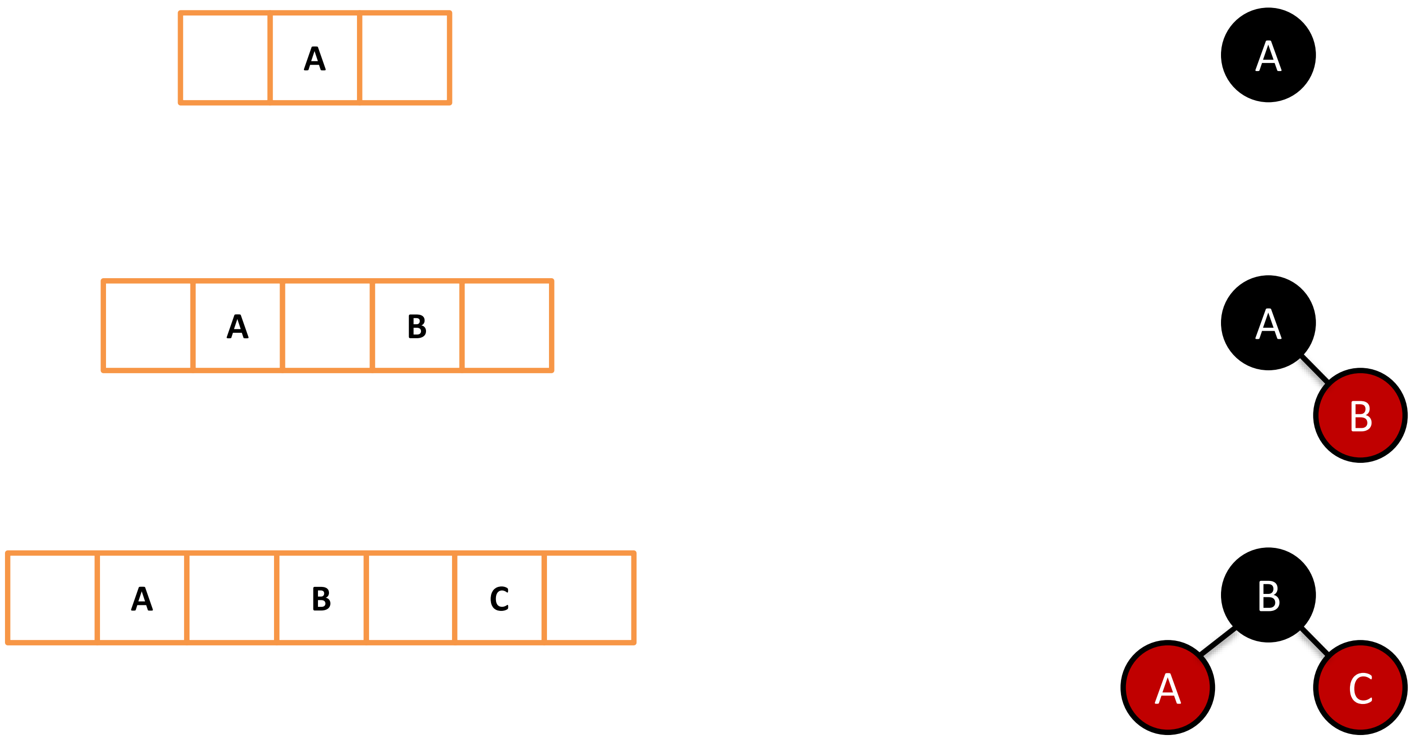
Promote node (C)
Rotate about parent



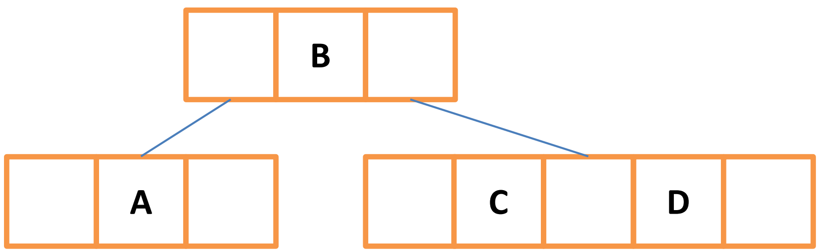
Promote node (C)
Rotate about parent

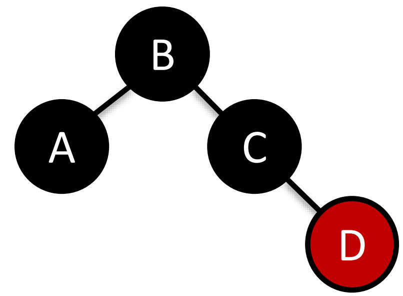


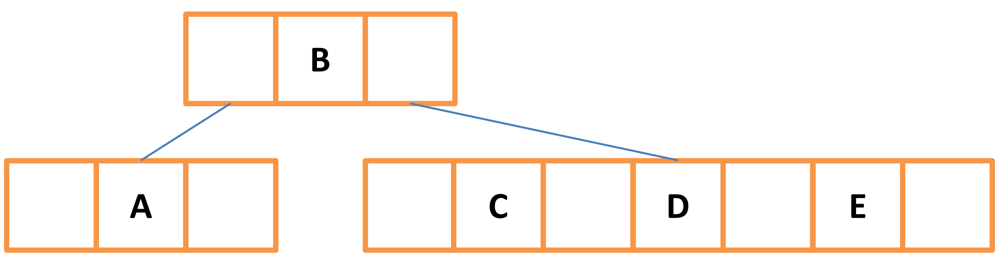
Build the trees in the order “A, B, C, D, E, F, G, H”

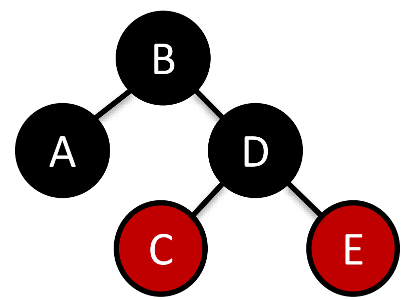


Build the trees in the order “A, B, C, D, E, F, G, H”

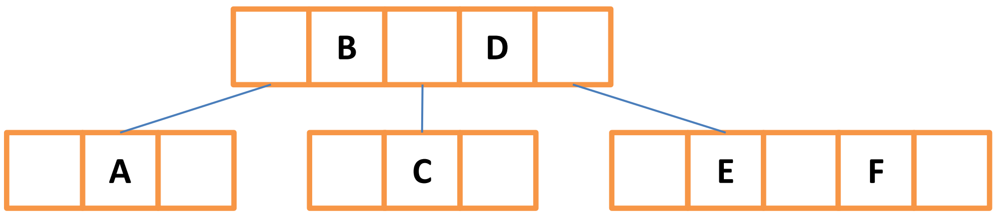


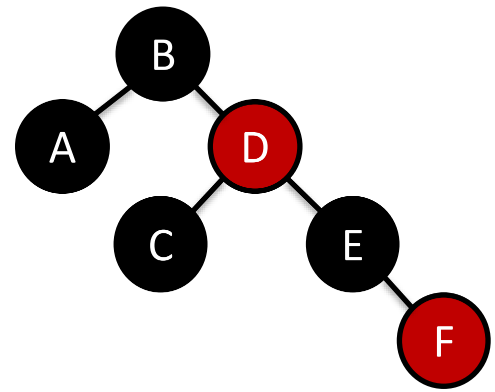


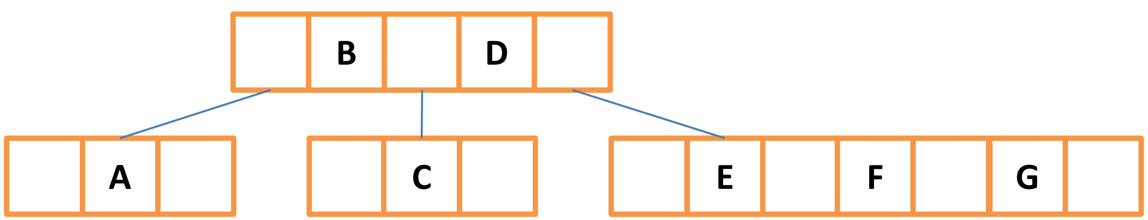


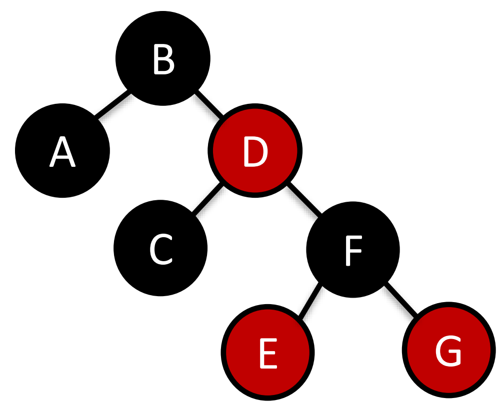


Build the trees in the order "A, B, C, D, E, F, G, H"

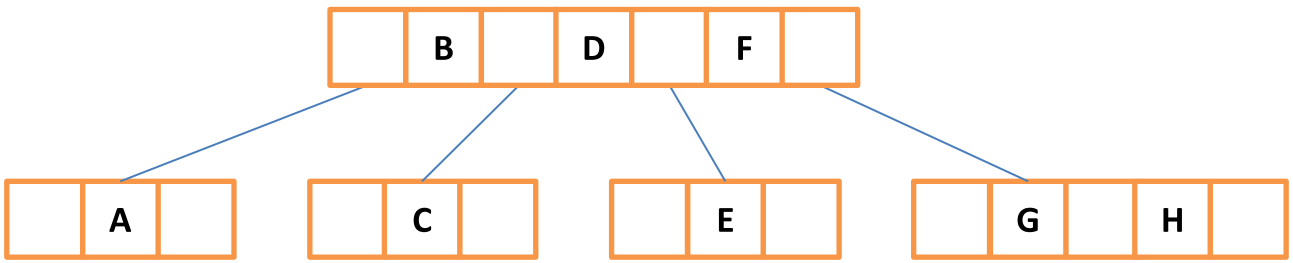


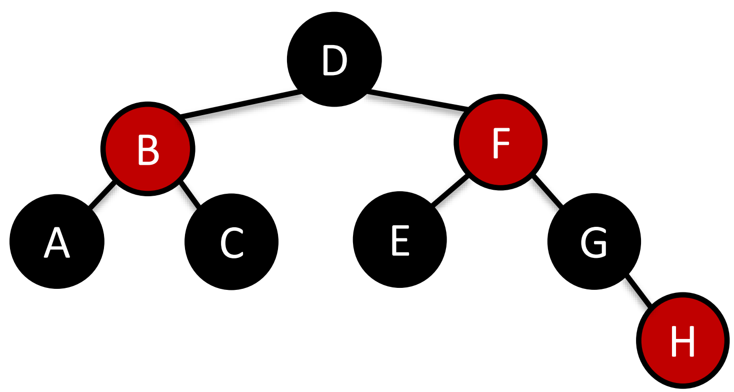






Build the trees in the order “A, B, C, D, E, F, G, H”



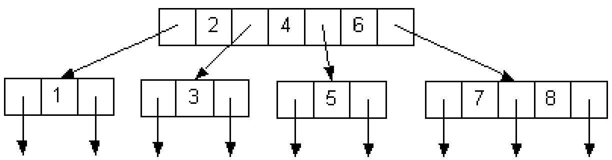


Practice

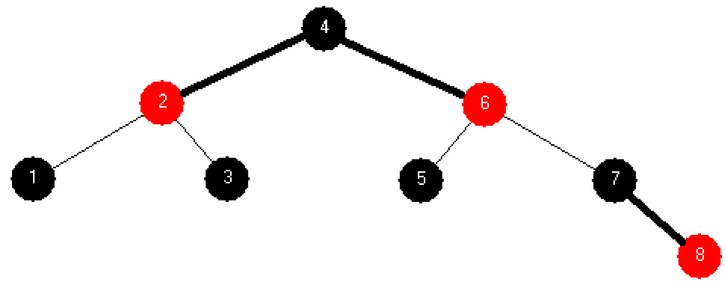
- Build a 2-3-4 tree and a R-B tree with
- 11, 2, 12, 1, 7, 3, 5, 8, 4, 6, 9, 10

2-3-4 Tree vs RB Tree vs BST

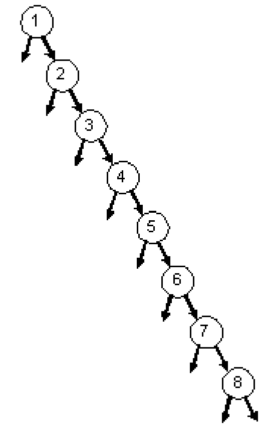
2-3-4 Tree: 1 2 3 4 5 6 7 8



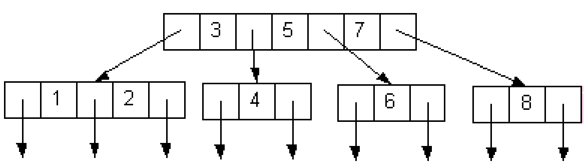
Red-Black Tree: 1 2 3 4 5 6 7 8



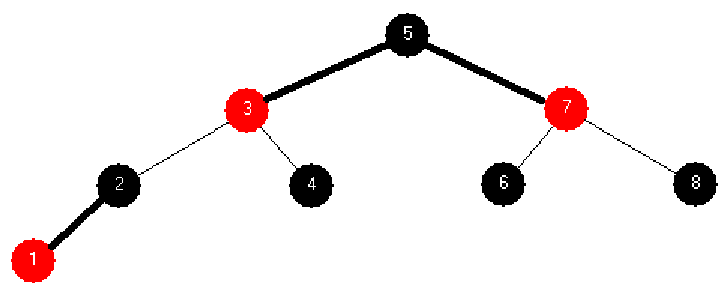
BST: 1 2 3 4 5 6 7 8



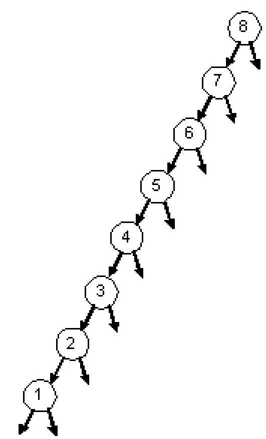
2-3-4 Tree: 8 7 6 5 4 3 2 1



Red-Black Tree: 8 7 6 5 4 3 2 1

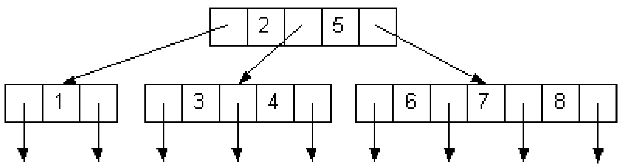


BST: 8 7 6 5 4 3 2 1

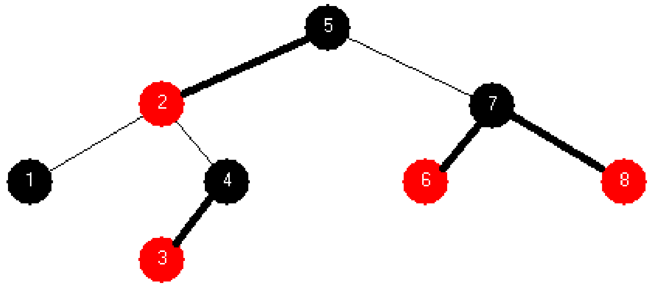


2-3-4 Tree vs RB Tree vs BST

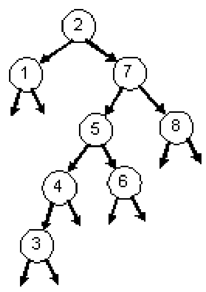
2-3-4 Tree: 2 7 5 6 1 4 8 3



Red-Black Tree: 2 7 5 6 1 4 8 3



BST: 2 7 5 6 1 4 8 3



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.
- The insertion/rebalancing algorithm is fairly simple.
- Searching, inserting, and re-balancing are all **O(log N).**

Summary

- 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 $\log_2 \mathrm{N}$ comparisons to reach a leaf. The worst case complexity, then, is O(log N).
  2. The Red-Black tree is approximately balanced and requires at most 2 log N comparisons to reach a leaf. The worst case complexity, then, is O(log N). On average, the number of comparisons is $1.002 \log_2 \mathrm{N}$.