# PG4200: Algorithms And Data Structures

# Lesson 05:
# Tree Maps

Dr. Andrea Arcuri

# Overview

- Together with lists, *maps* are the most common data structures used in programs

- No "index" of position, in contrast to arrays/lists

- Association between "**keys**" and "**values**"
  - Ie keys are *mapped* to values

# Insert/Find

- When inserting a new *value V*, also insert a *key K* for it

- A K key is an object (eg, String, Integer, etc.)

- When we want to retrieve V, we will query for its K

- The idea is that retrieving data for given K should be *fast*

# Examples

- Collection/map of songs: the Song class is the V, where the K is the String title

- Collection/map of users: the User class is the V, where the K is the Integer id of the user

# Search

- How can we efficiently search for a K?

- Let's assume that Ks are orderable, eg a *compareTo()* is defined for them

    - This is the case for String and Integer

# Search Value in Array

| 7 | 2 | 4 | 9 | 1 | 3 | 0 | 5 | 8 | 6 | 10 |
|---|---|---|---|---|---|---|---|---|---|----|

- Assume content of array is numeric keys

- And we want to search for key 8

- Scan from left-to-right, and check if == 8

- Complexity O(n)

# Search In Sorted Array

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | **92** | 93 | 94 | 99 |

- Assume we search for 92

- Is there any clever way to check if 92 by exploiting knowledge of array being sorted?

# Binary Search

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |

- Check middle of array, ie 78

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |

- As 78 < 92, check middle of subarray between 80 and 99, ie 93

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |

- As 93 > 92, ignore the rightmost values

- Look at middle of 80-92

- Finally find 92

# Complexity

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |
|---|----|----|----|----|----|----|----|----|----|----|

- At each step, look at area that is half in size

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |
|---|----|----|----|----|----|----|----|----|----|----|

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |
|---|----|----|----|----|----|----|----|----|----|----|

- How many times *x* we can divide N by 2 till we arrive to value 1?

| 7 | 11 | 40 | 42 | 70 | 78 | 80 | 92 | 93 | 94 | 99 |
|---|----|----|----|----|----|----|----|----|----|----|

- $N = 2^x \rightarrow x = \log(N)$

- Binary Search has O(log N) complexity, which is much better than O(N)

# List/Array Based Map

- Each element is an Entry object, containing K and V

- **Insertion: O(N)**, can keep list/array sorted

- **Deletion: O(N)**

- **Search** for K: **O(log N)** if sorted, otherwise O(N)

- *Can we do better?* Eg, **O(log N)** in all operations

# Tree-based Maps

- Data structures resembling a tree

- Nodes contain values, and links to child nodes

- Starting point is the root of the tree

- Two main / most famous versions

  - Binary Trees ("simple")

  - Red-Black Trees ("very complex", but high performance)
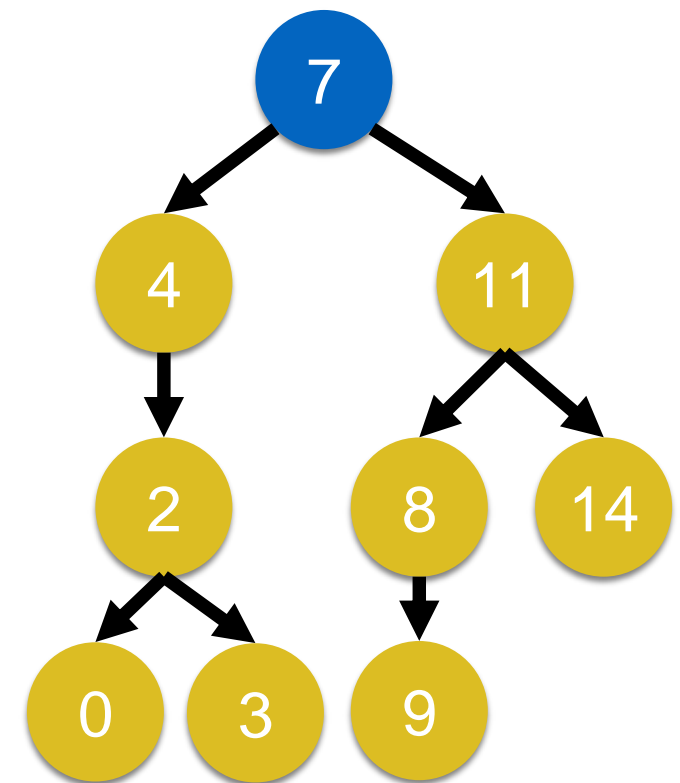
# Binary Tree

- Each Node has 1 value and up to 2 children: *left* and *right*

- Values in *left* subtree are all *smaller*

- Values in *right* subtree are all *larger*

- Insertion/deletion do keep these invariants

- Search starts from root
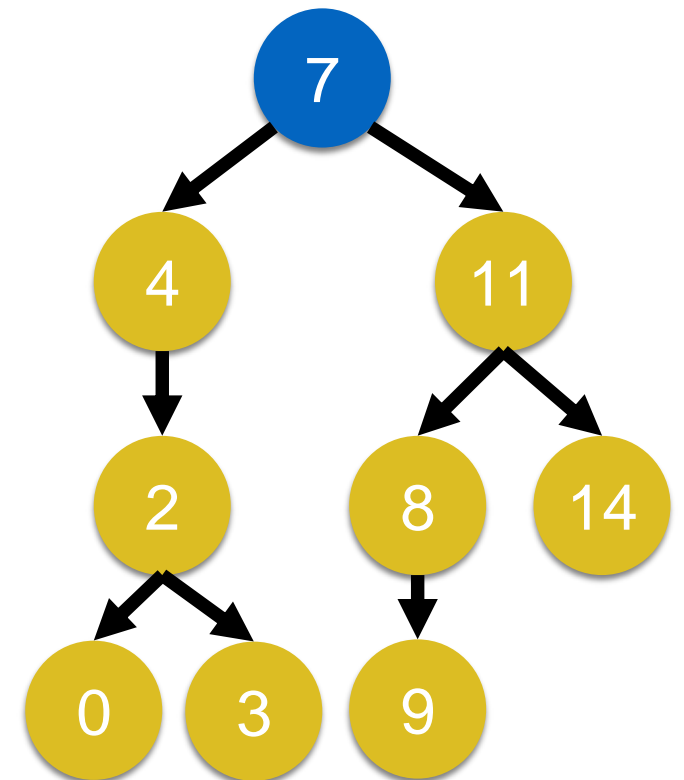
# Insertion

- Assume inserting entries with keys 4, 2, 3, 5 and 1

- Recall: each node has a key and an associated value

- Here in the circles I am just showing the keys, not the values
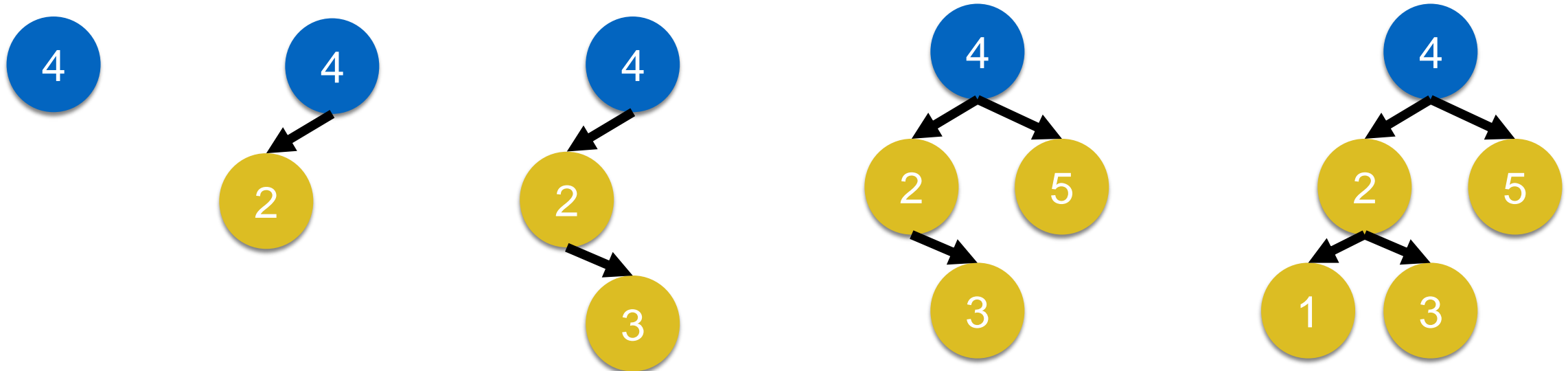
# Insertion Order

- Assume inserting same entries but in this order: 5, 4, 3, 2, 1

- Shape of tree depends on order in which elements are inserted

- Worst case the tree degenerates into a list
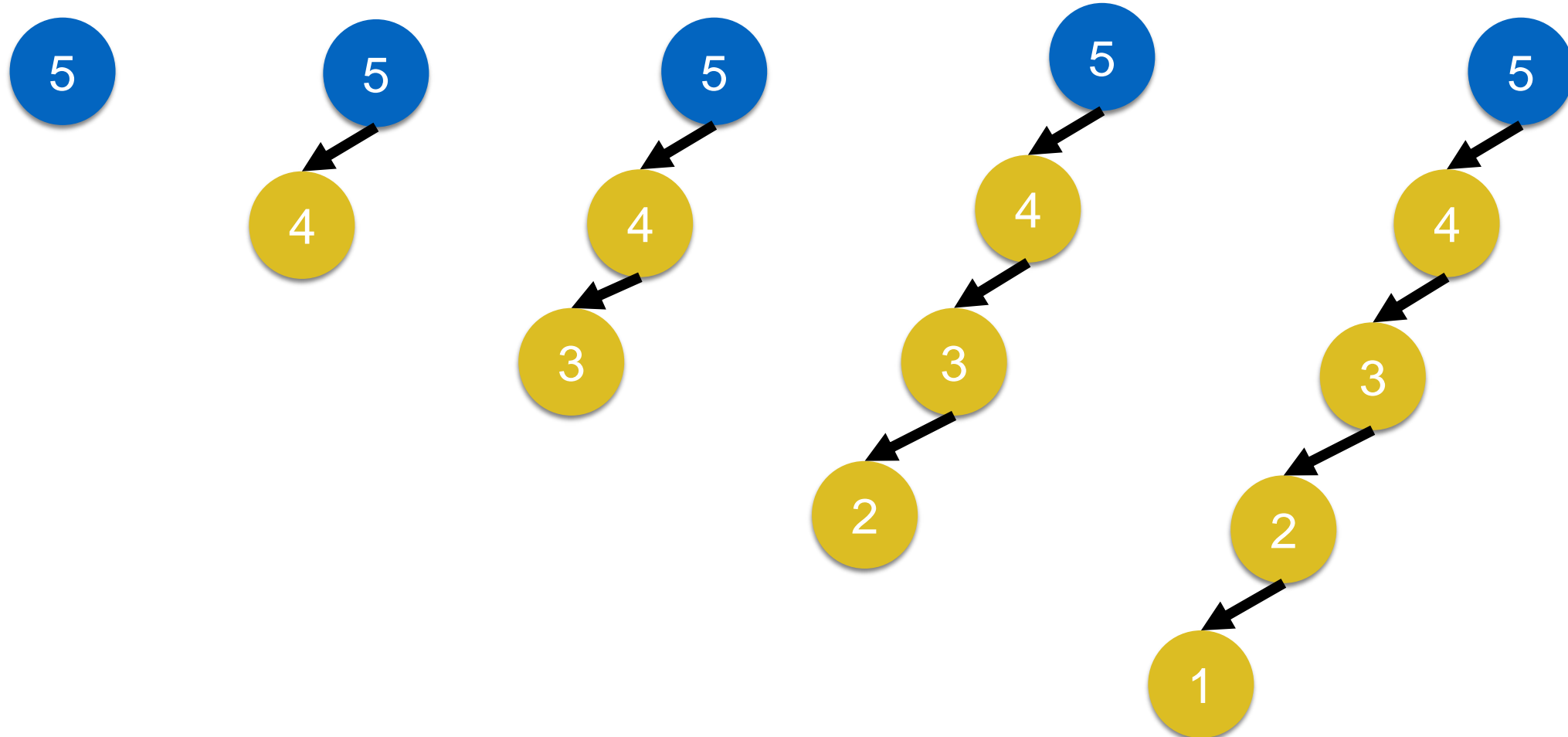
# Search

- Start from root

- Recursively navigate left or right subtree based on when searched key is smaller or bigger than the one in the node

- Worst case: searched key is in a leaf node

- Max number of comparisons bound by *depth* of tree

  - Eg 4 in the example
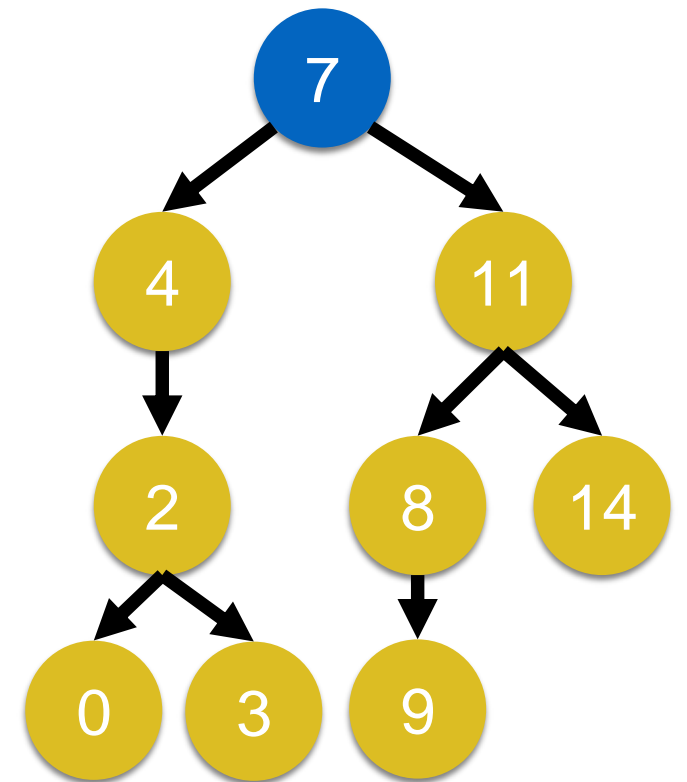
# Tree Depth

- Depth: longest distance between root and a leaf

- Given N nodes

- Depth bounded between N (worst case) and log(N) (best case)
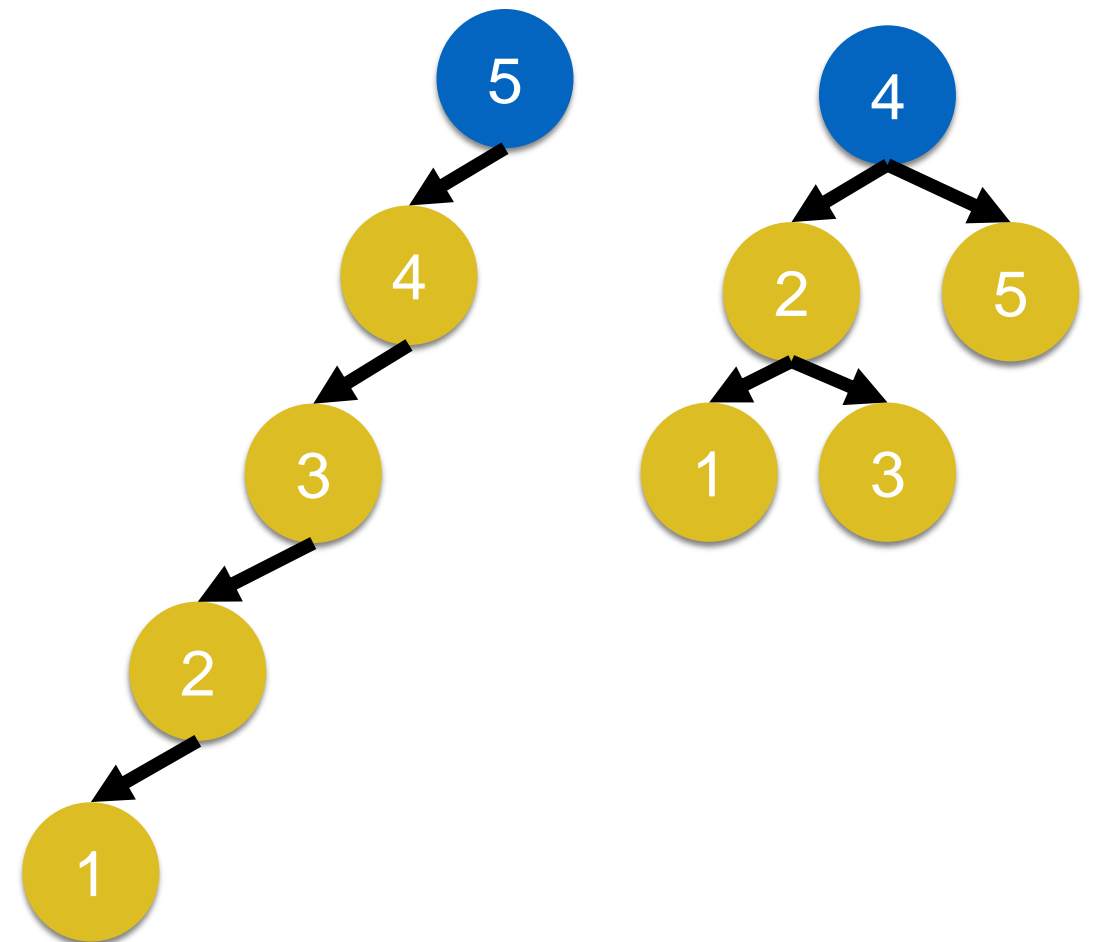
# Deletion

- To delete K, need to find it first

- Actual deletion depends on numbers of children of K

- Simple case: deleting with no children, eg delete key 9, just set to null the link of the parent (8 in this case)
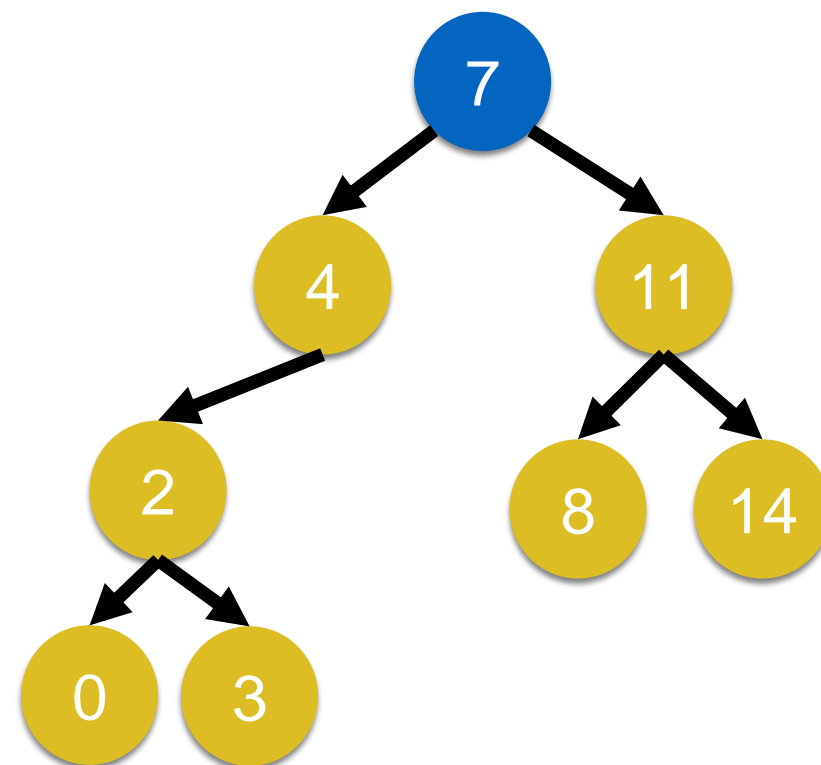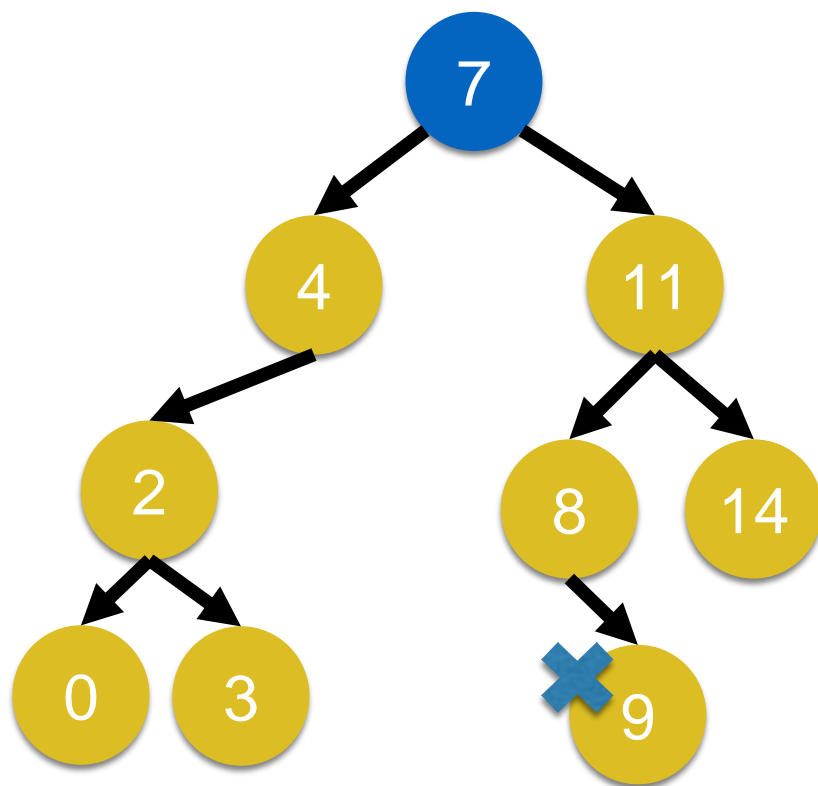
# Deletion With 1 Child

- Eg, delete 8 (or 4)
- Look at the parent of deleted node (ie 11)
- Change the child link of parent to the child of deleted node (ie 9)
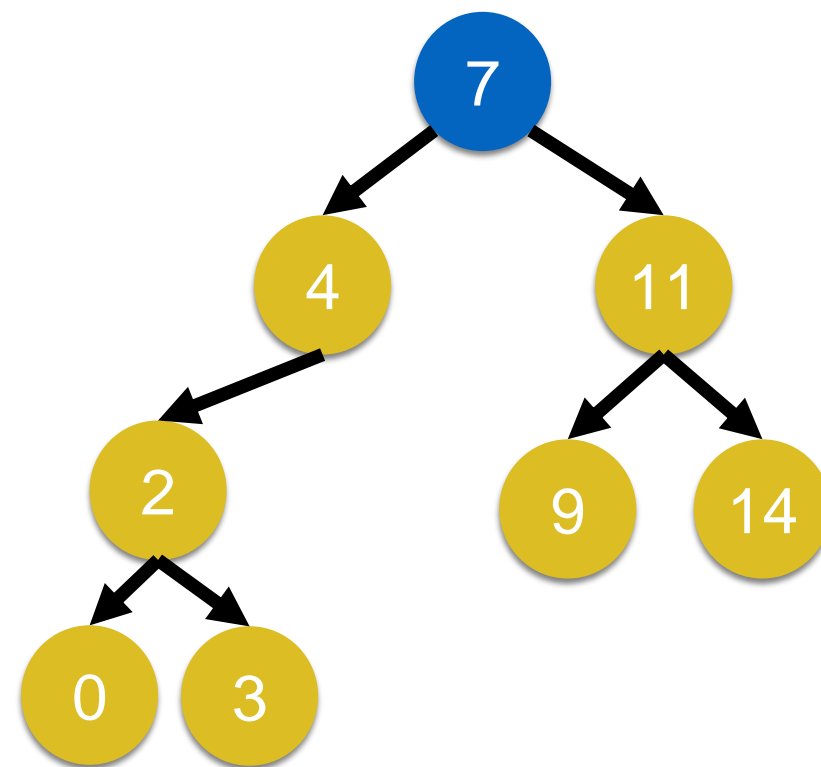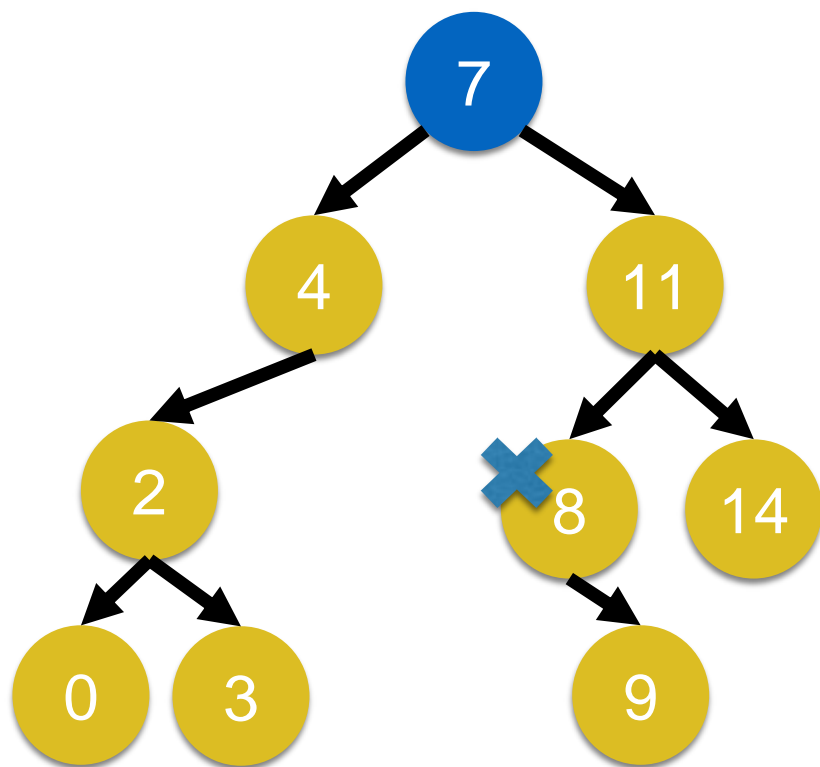
# Deletion With 2 Children

- Eg, delete 7 (or 2 or 11)
- Find *minimum* in right subtree (ie 8) and *delete* it
- Replace the content of right-minimum into deleted node
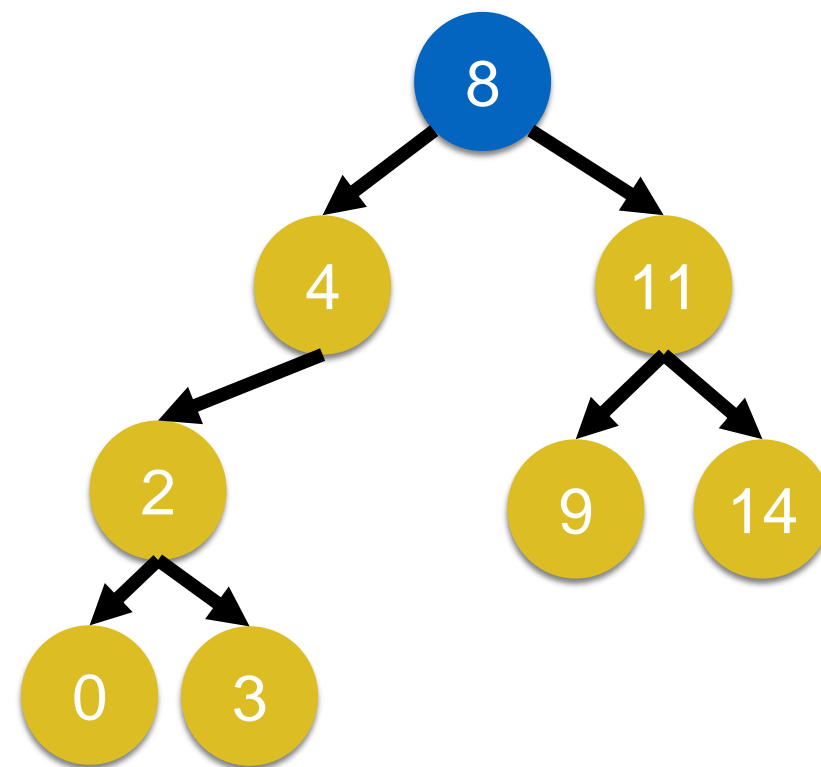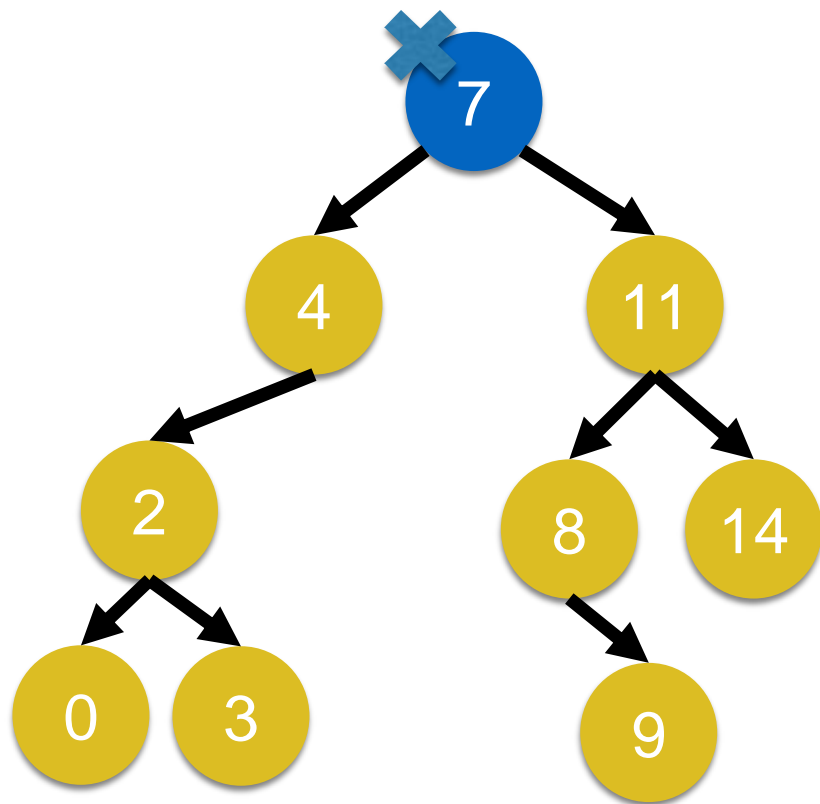
# Runtime Complexity

- In worst case, Binary Map degenerates into a list

- All operations have **O(N)** complexity in worst case

- But random trees would be nearly balanced, and operations would hence be in the O(log N) order

  - So good performance on average

# Red-Black Trees (RBT)

- Complex type of tree
  - Likely the most complex algorithm we see in this course (and likely in your degree as well…)
  - If you have difficulties in this course, keep this one as last to study/revise, unless you are aiming for an **A** grade…

- Insertions and deletions *always* keep the tree balanced

- Guarantee **O(log N)** on all operations
  - Ie, insertion, deletion and search

- Extremely efficient
  - *Can **guarantee** search into **B**illions of data with just 30-50ish node lookups*

# 2-3 Search Trees

- Before discussing RBT, let's consider 2-3 Trees

- 2-3 Tree: composed of 2-nodes and 3-nodes

- 2-node: 1 value, 2 children (left and right)

- 3-node: 2 values, 3 children (left, middle and right)
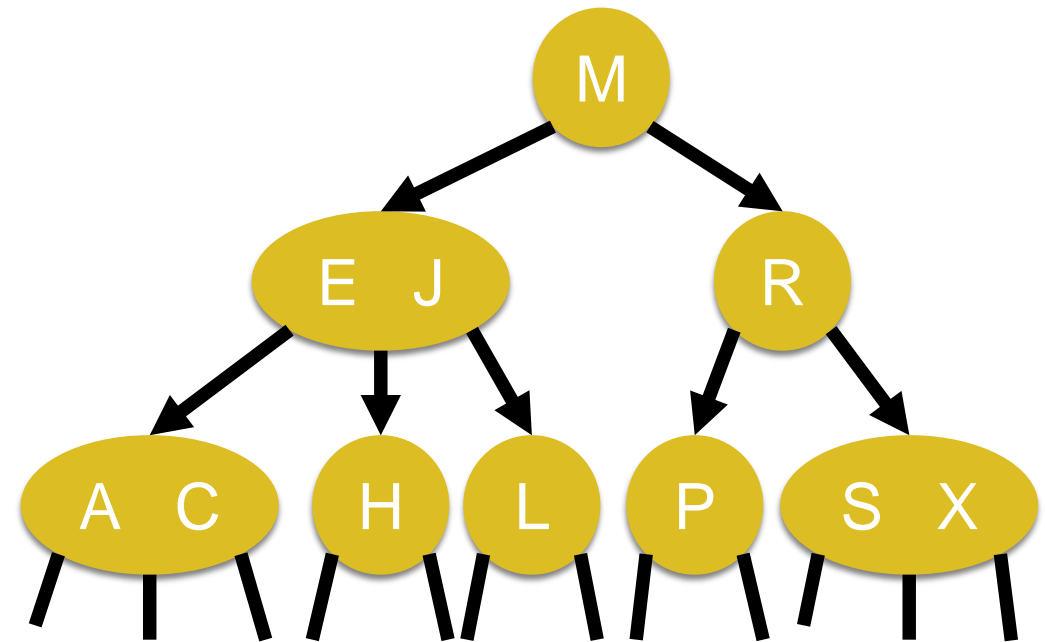
# Perfectly Balanced Trees

- When distance from root to any null-child-link is the same
  - Important property to guarantee O(log N)
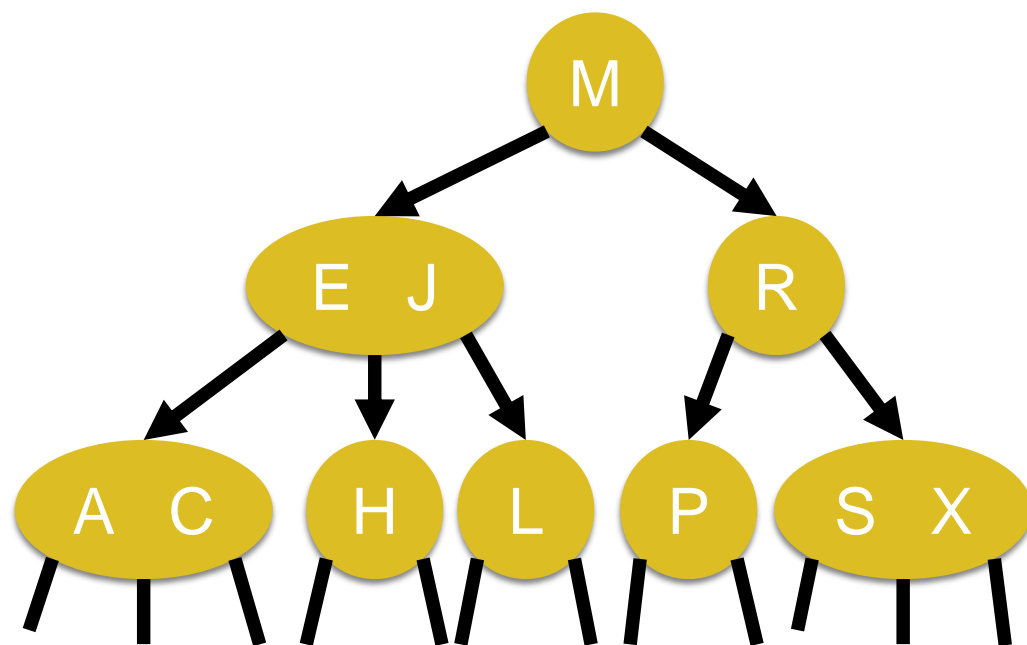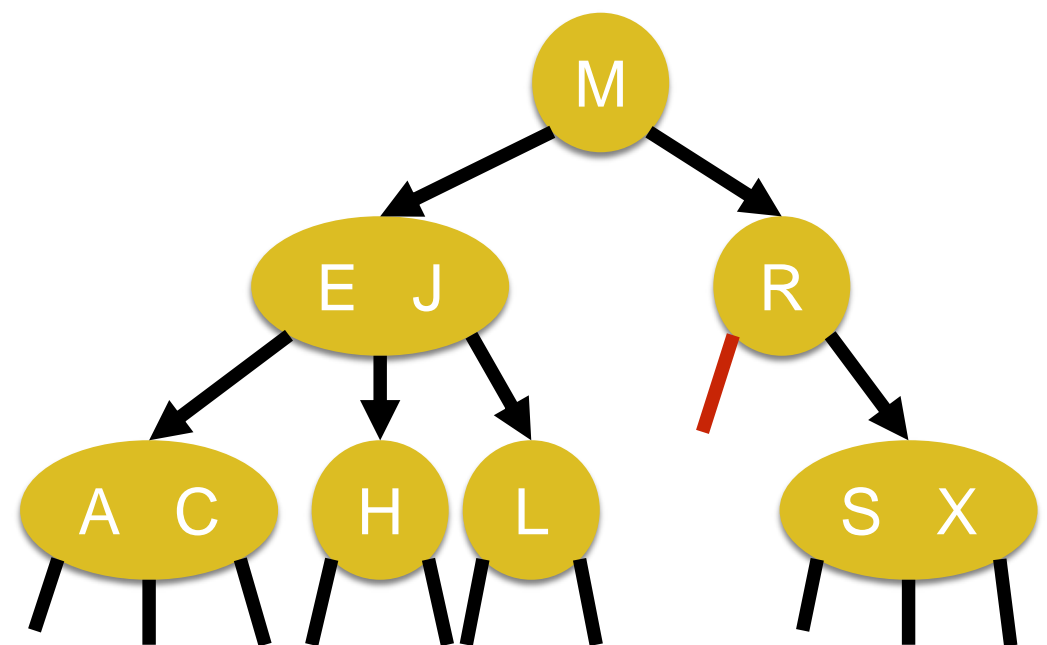


Balanced

Not Balanced

# Insertion

- After an insertion we want to keep the tree balanced

- Based on key, search starting from root to the position where to insert
    - This takes O(log N) steps
    - Same concept as Binary Search Trees

- Insertion will depend on position, but many cases to consider

- Insertion on empty tree: just create a 2-node root

# Insertion on 2-node

- Trying to insert **K**, reached node for L
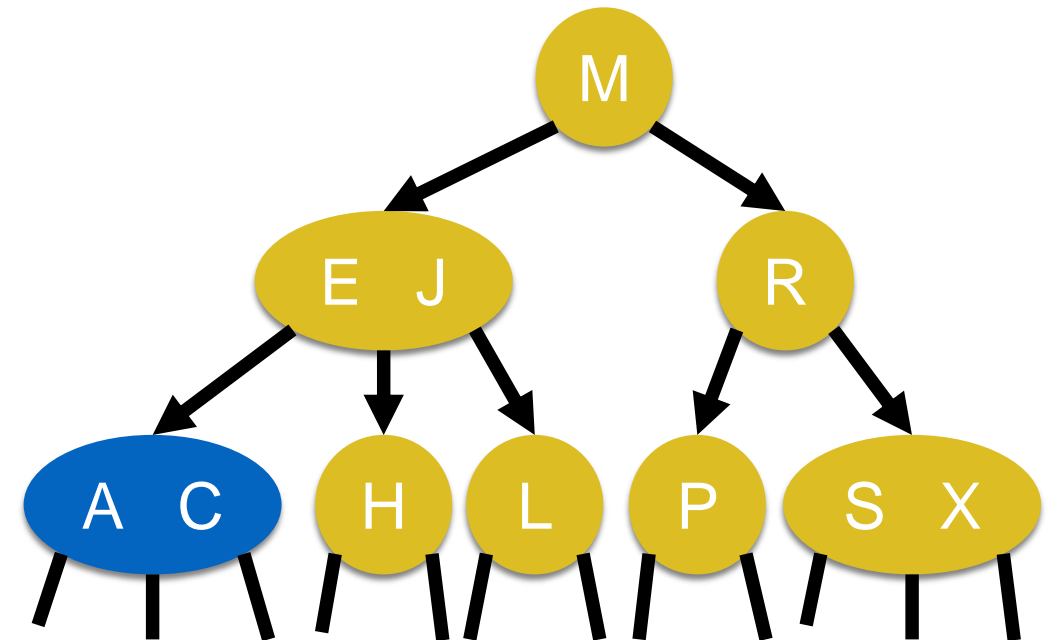
- Node for L is a 2-node: easy, just transform into 3-node
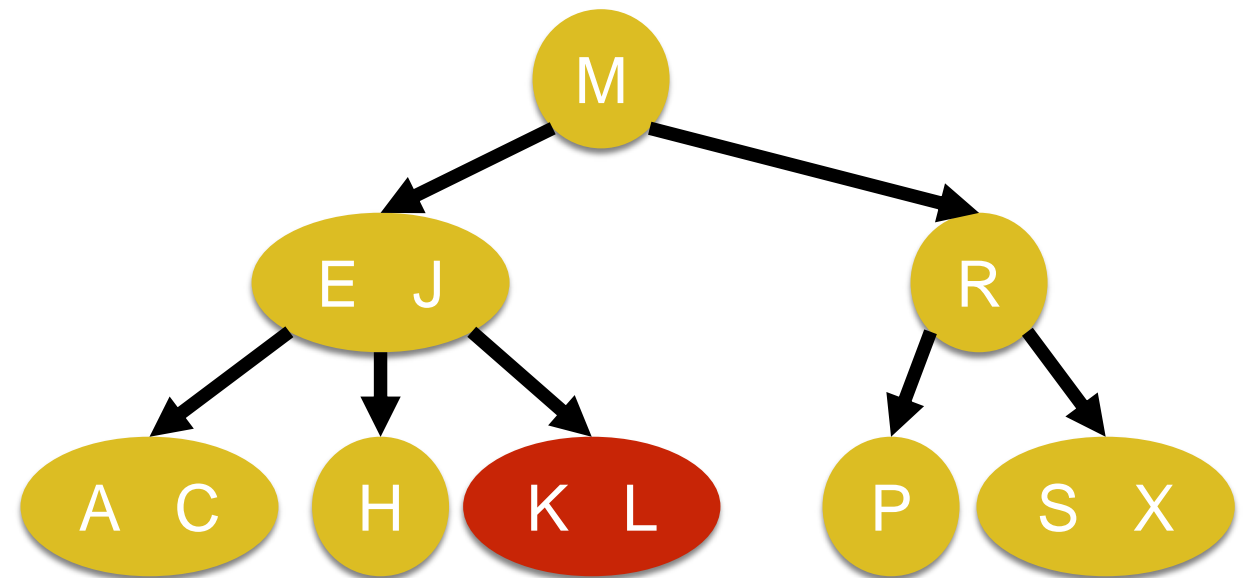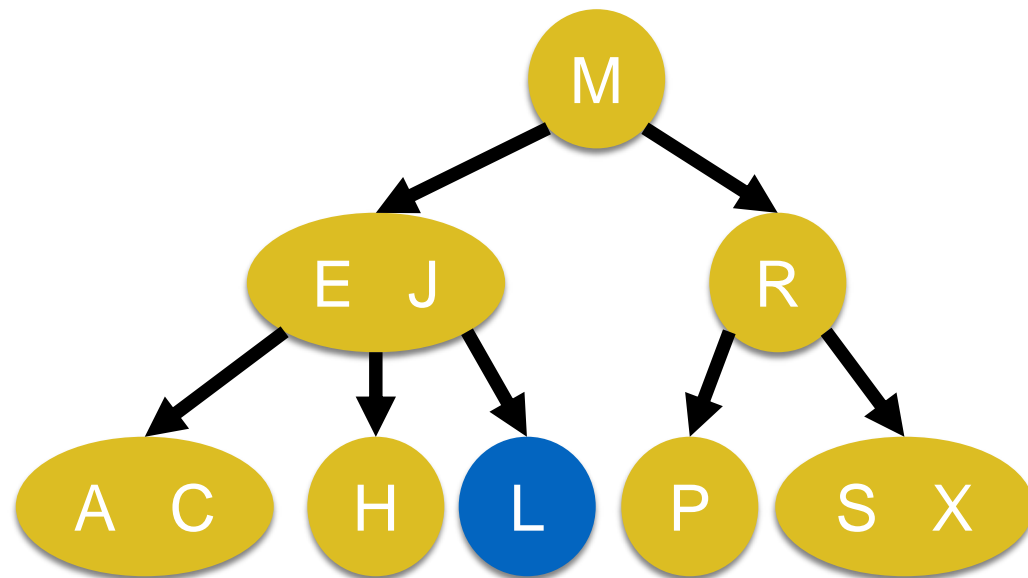
# Insertion on 3-node, where Parent is a 2-node

- Inserting **Z**
- 3-node (S,X) with 2-node parent (R)
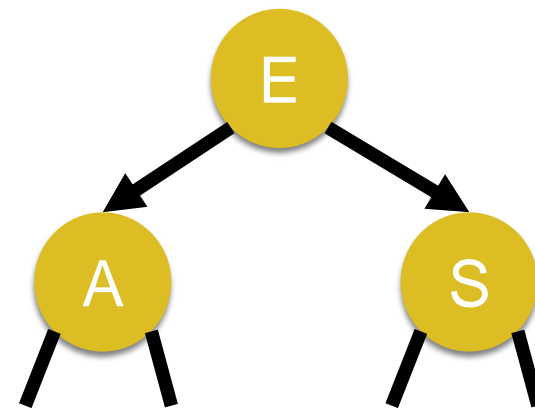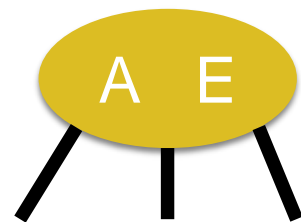- Split node, and transform parent into a 3-node
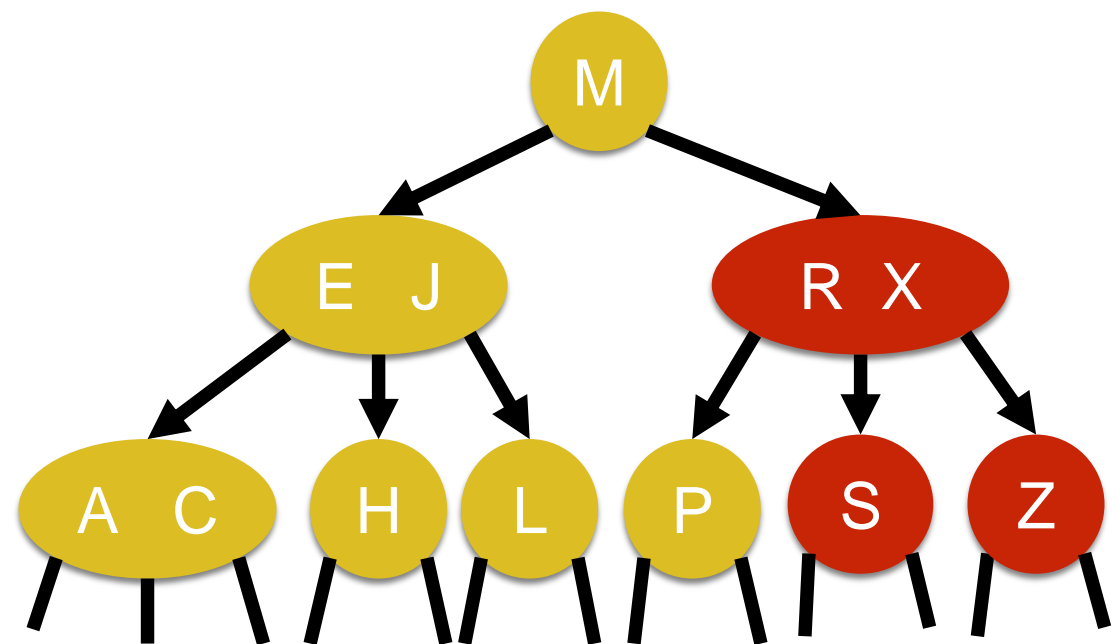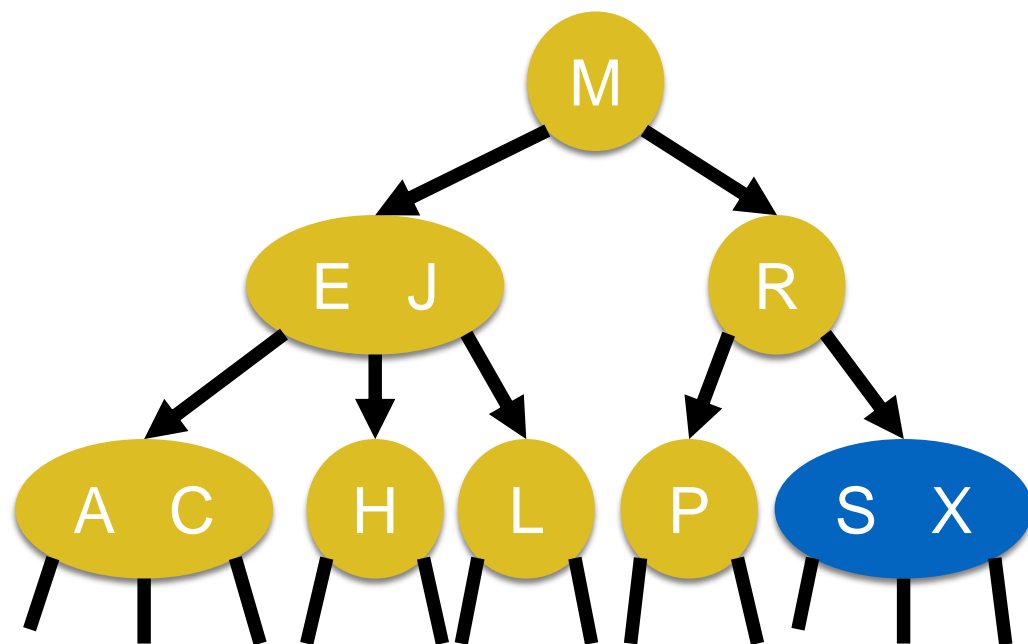
# Insertion on 3-node, where Parent(s) are 3-nodes

- Inserting **D**

- 3-node (A,C) with 3-node parent (E,J)

- Split node, add middle to parent, repeat recursively bottom up until find 2-node
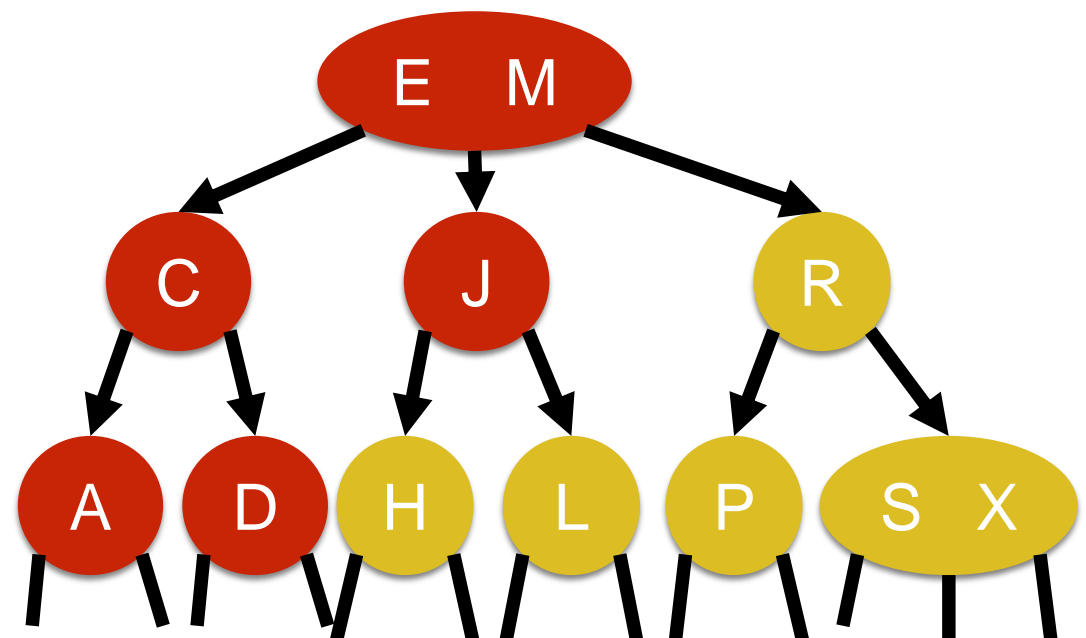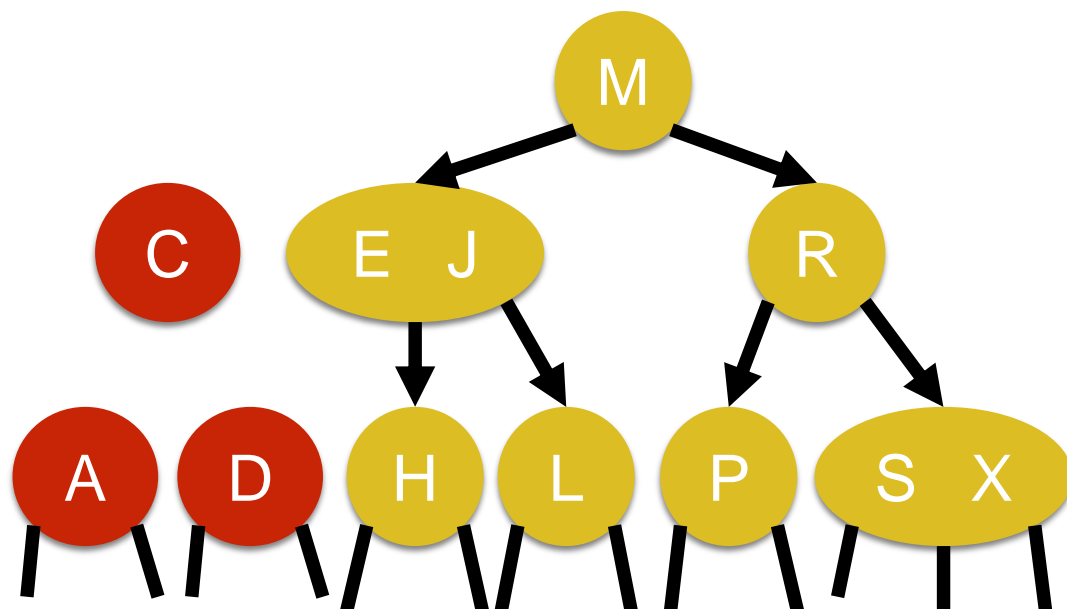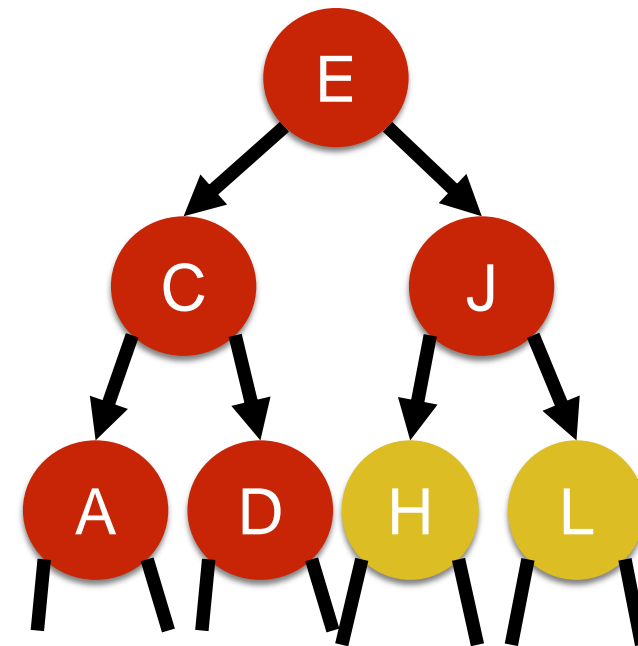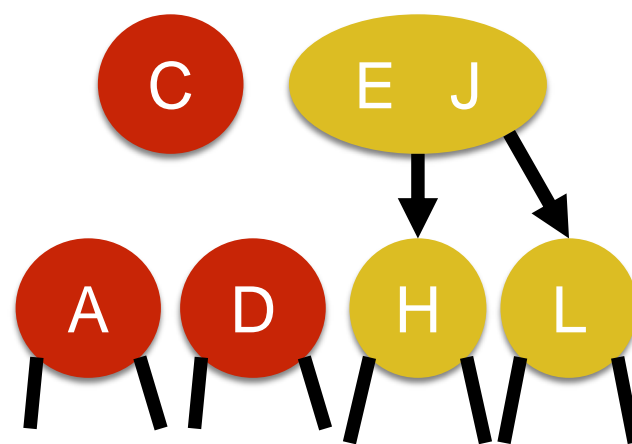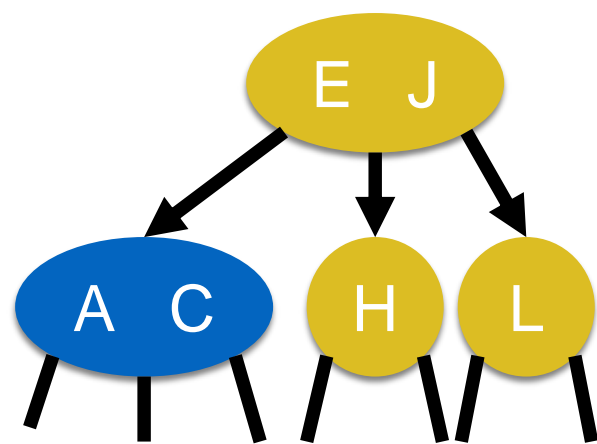
# Insertion on 3-node Root, with No Children

- Inserting a new value, eg **S**

- Split the 3-node, and create new 2-node

# Splitting The Root

- Adding **D**
- What happens when going bottom-up adding recursively to parent we reach a 3-node root?
- Same as root with no children
- Note: this is the only case in which depth increases

# Cost

- Searching for position to insert is O(log N)

- Worst case, when inserting, need to go upward till the root, which add a further O(log N)

- So final cost of insertion is O(log N)

  - Recall, constants are not important in asymptotic analyses, so O(log N) + O(log N) = 2O(log N) = O(log N)

- After each insertion, tree is perfectly balanced

# From 2-3 Trees to RBT

- Using 2-3 trees is cumbersome, as we need to handle both 2-nodes and 3-nodes

  - And not particularly efficient when transforming 2-nodes into 3-nodes, and vice-versa

- RBT: represents 2-3 Trees by only using 2-nodes

- Each link between nodes has a color: RED or BLACK

  - This is where the RBT name comes from

# 3-nodes

- Represented with two 2-nodes
- Link with smaller is RED, and RED links can only lean left
- Any other kind of link is BLACK
- For simplicity, using color directly on the node to represent link to parent
- Cannot have two consecutives RED links
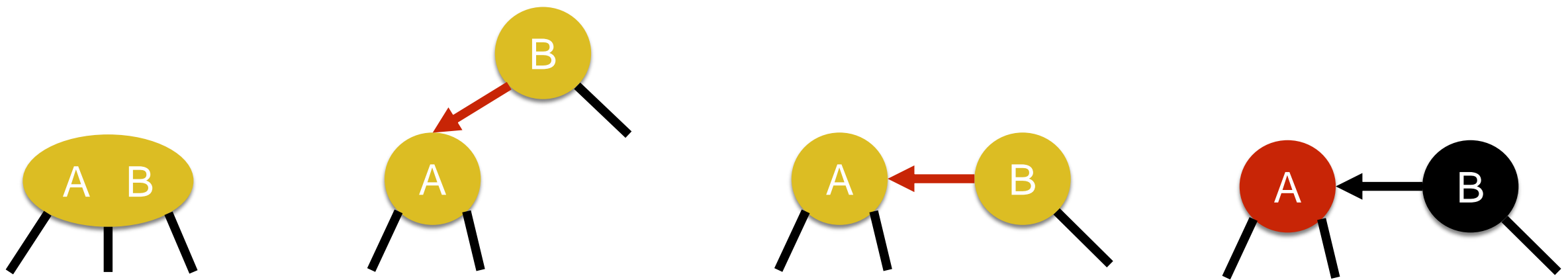
# Black-links Balance

- In RBT there is only guaranteed balance for the black-links

- Every path from root to a black-null-link has same number of black links

- So, depth is still bounded by O(log N)

# Rotations

- When inserting/deleting, need to maintain RED-link properties, ie, most 1 leaning left

- So need to have transformation operators on the tree, which are used to maintain the tree balance
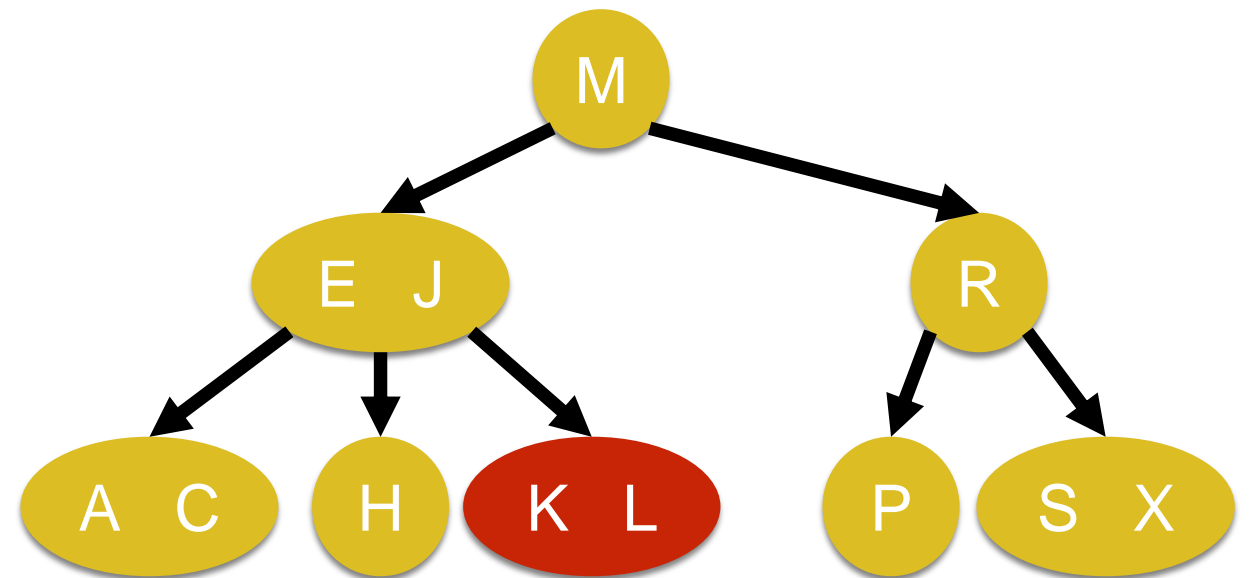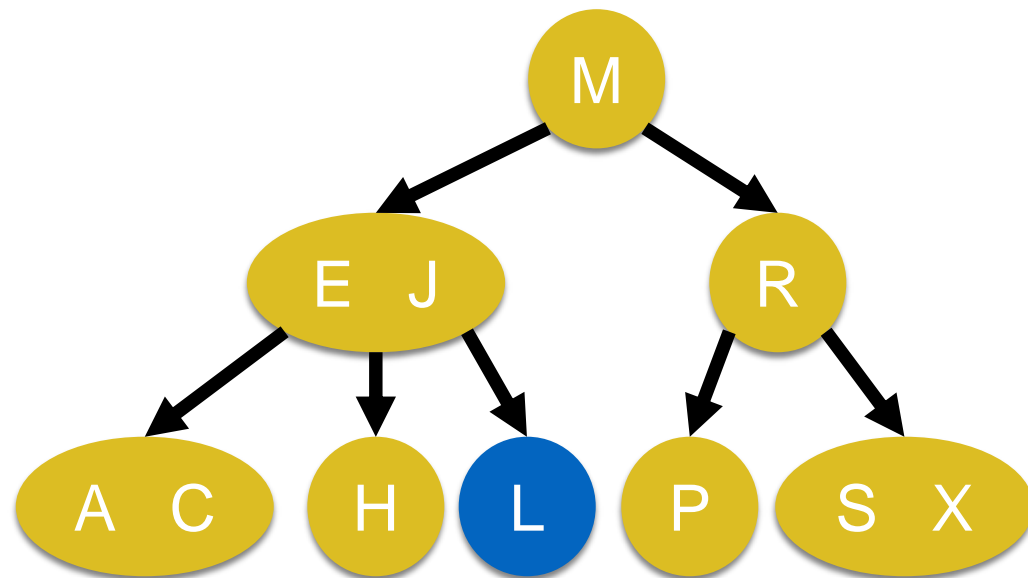
- These operations have O(1) cost

# Left Rotation

- S is moved up
- Still RED link between E and S, but now E is child of S
- Left child of S now becomes right child of E
- Color (RED/BLACK) of link toward E is maintained in new rotated root S
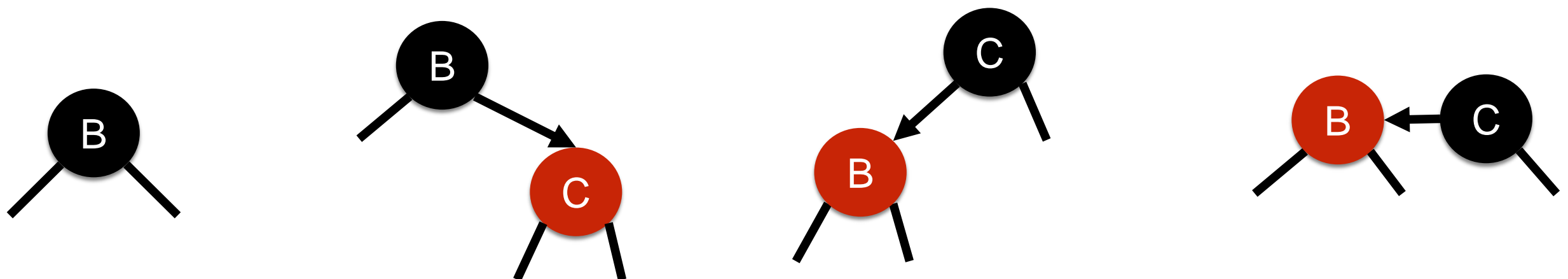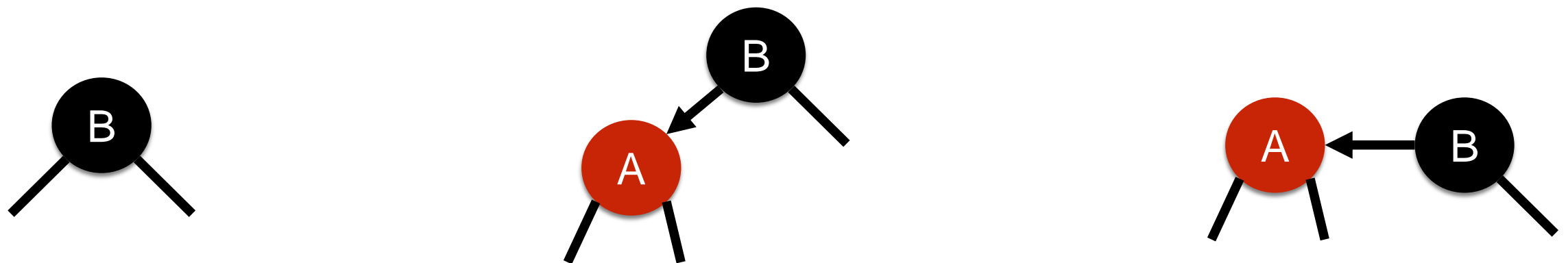- Right-rotation does the exact opposite
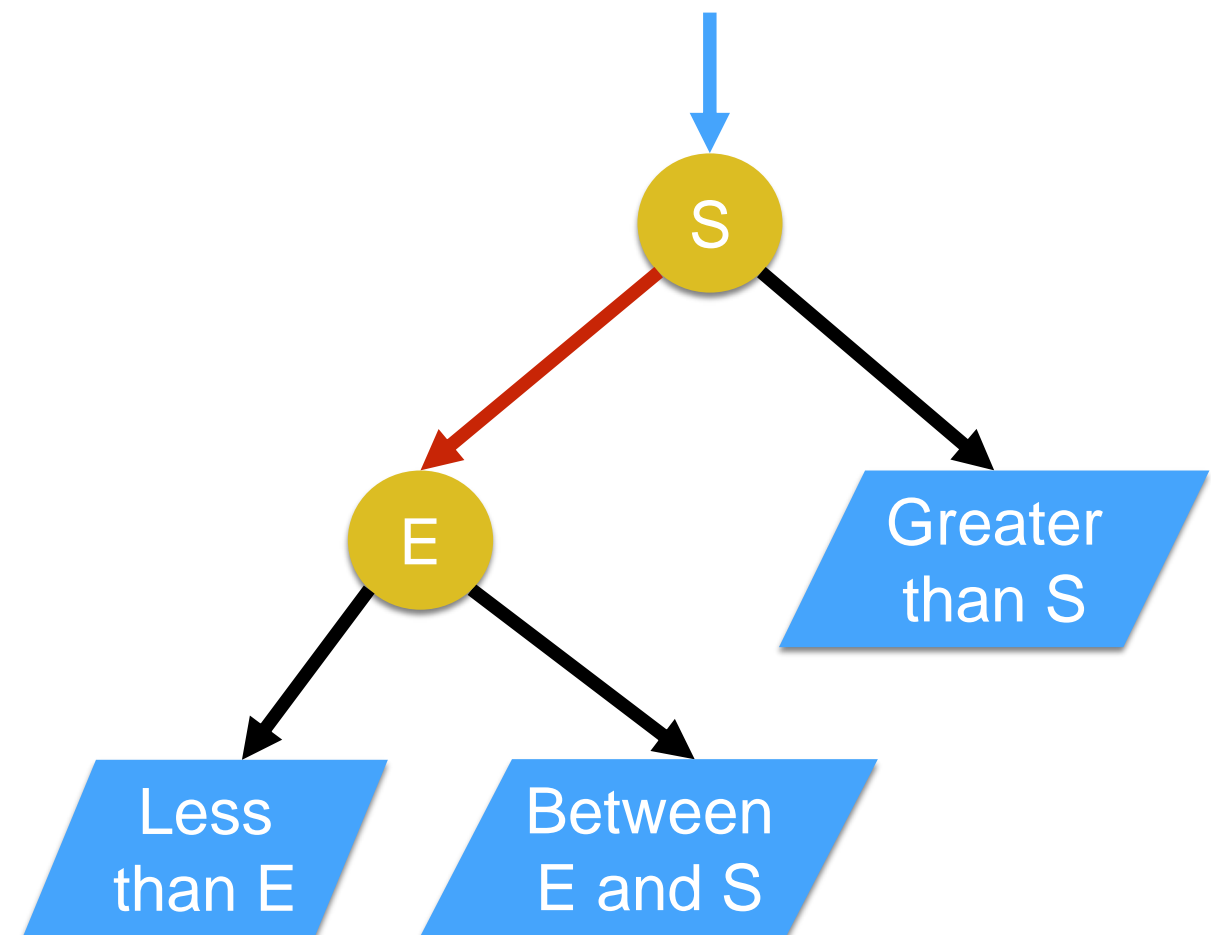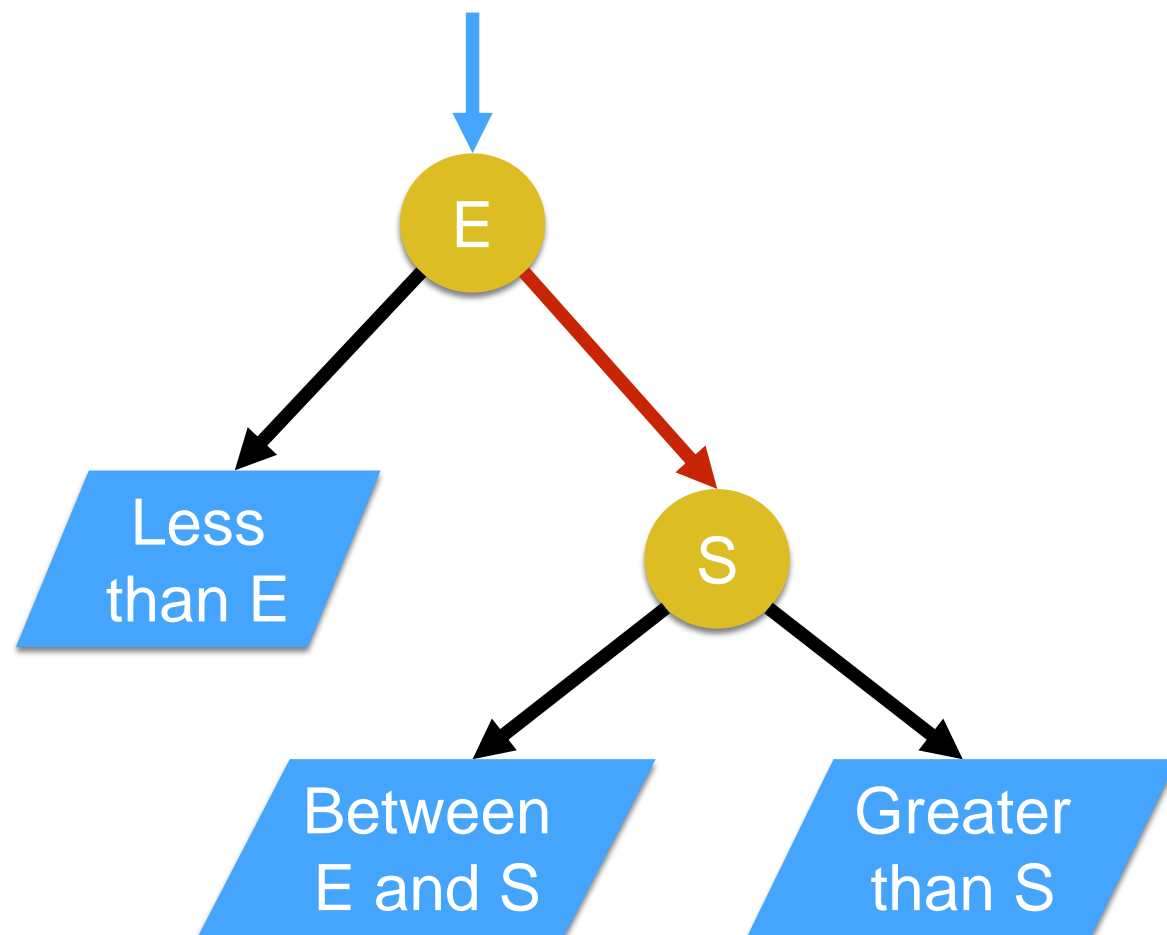
# Recall Insertion on 2-node
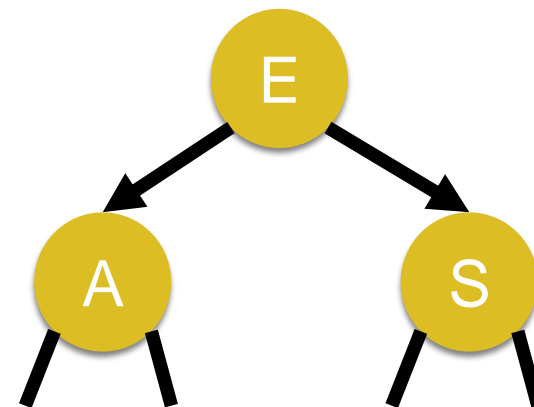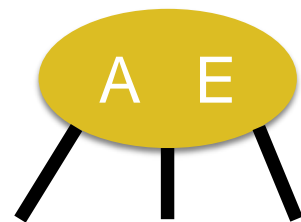
- 2-3 Trees

- 2-node transformed into 3-node

# Insertion into 2-node

- 2-node will need to transformed into a 3-node
- In RBT, need to have new node with RED link
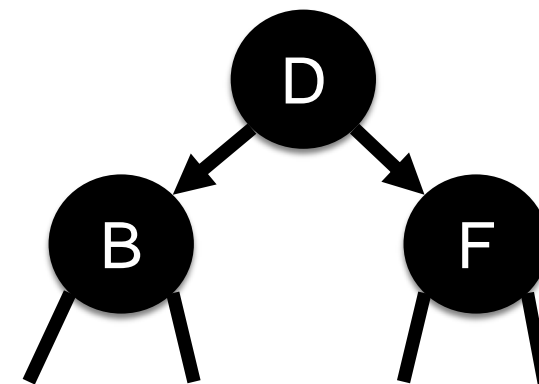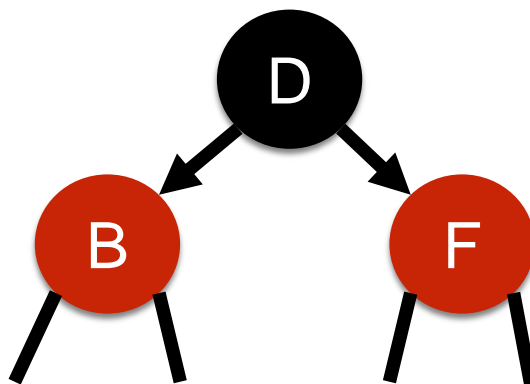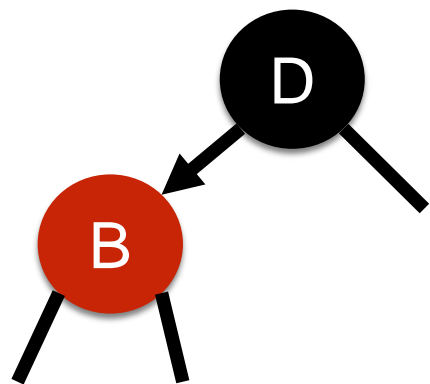- Need left rotation if new value is greater

# Recall Insertion on 3-node Root, with No Children

- 2-3 Trees

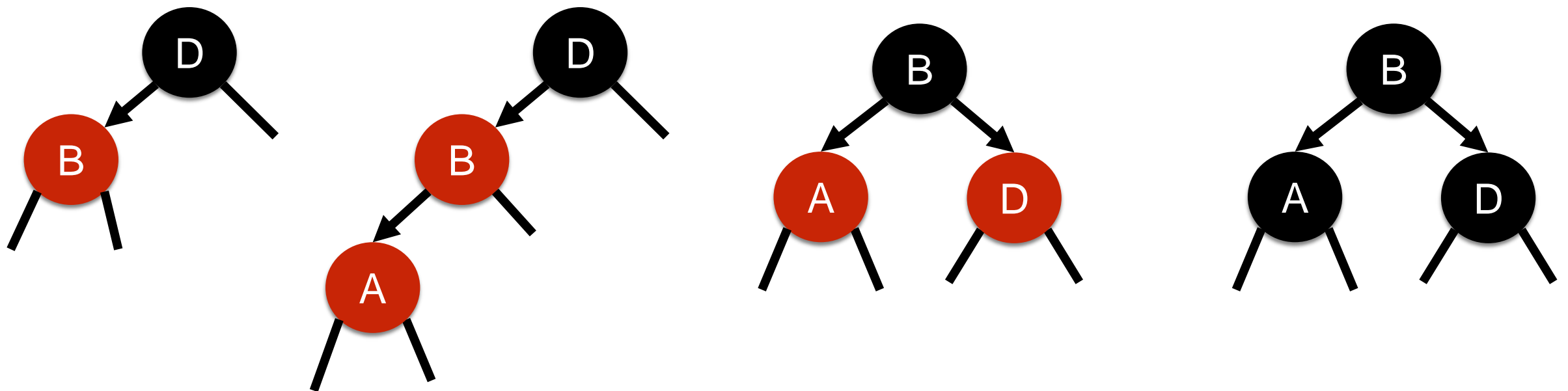- Split the 3-node, and create new 2-node

# Insertion in Single 3-node

- 3 cases, based on whether new value is larger, smaller or between

- Simple case: new value is greater, eg **F**

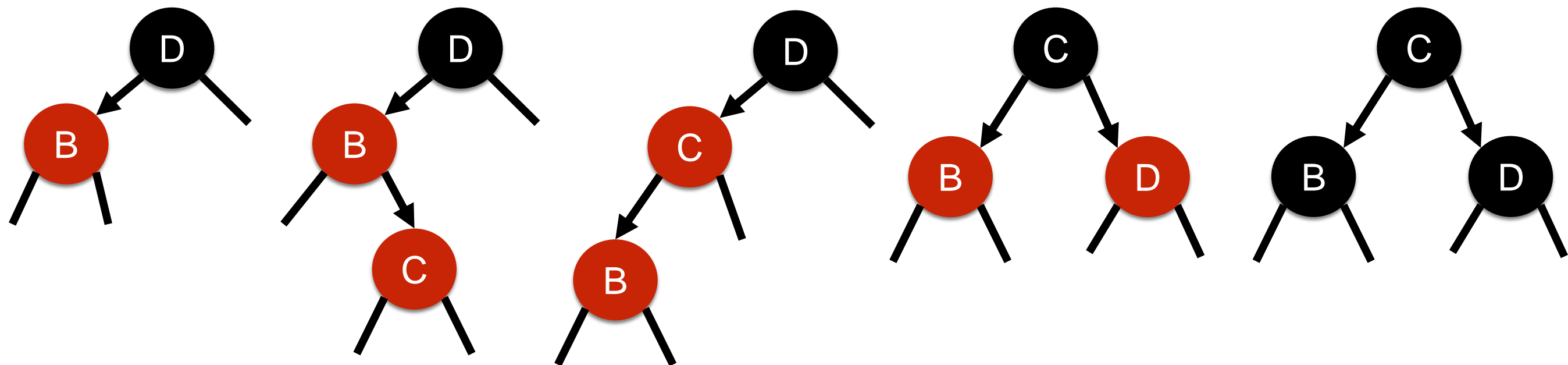- Flip color from RED to BLACK once inserted

# Insertion in 3-node: Smaller Case

- Smaller value, eg adding **A**

- After insertion, need right rotation

- Finally, need flipping of RED colors

# Insertion in 3-node: Between Case

- Between value, eg adding **C**
- After insertion, need left rotation, followed by right rotation
- Finally, need flipping of RED colors

# Cost of 3-node Insertion

- 3 cases, based on inserted key

- Larger: color flip

- Between: left rotation, right rotation, color flip

- Smaller: right rotation, color flip

- These are constant costs

# Insertion in Leaf 3-Node

- Flipping color makes parent RED
- Root is always put back to BLACK once insertion completed
- Inserting **H**, which "smaller" value in the 3-node (R,S)



Right rotation:
R and S

Color flipping:
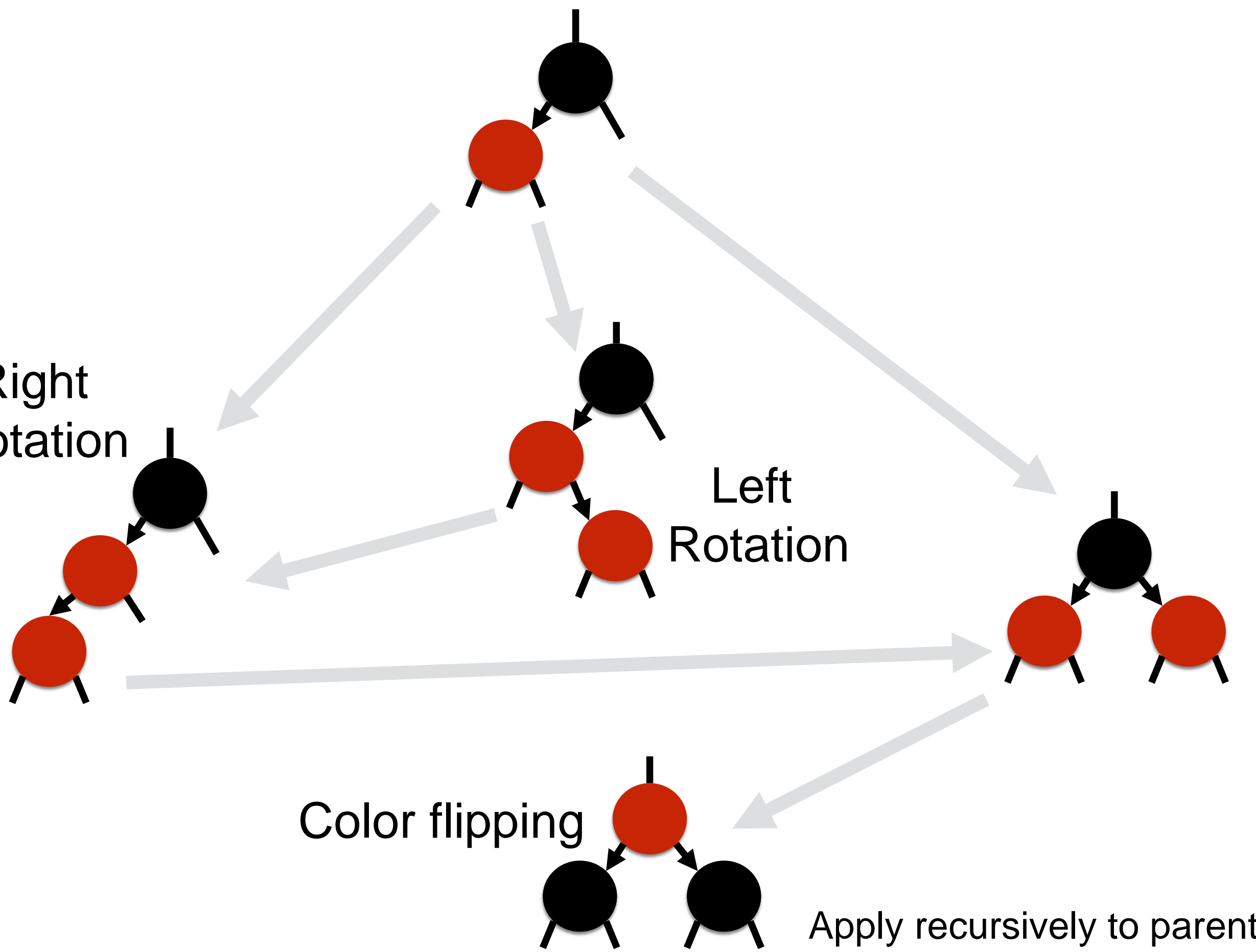H,R, and S
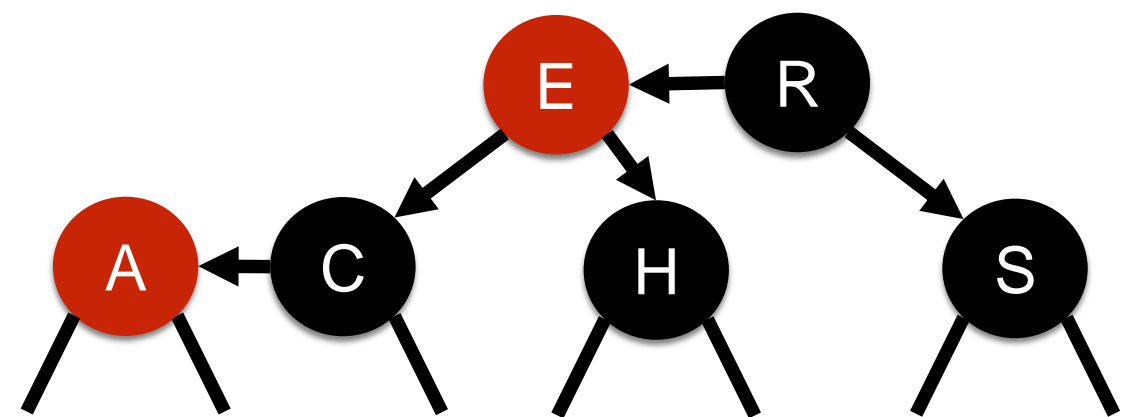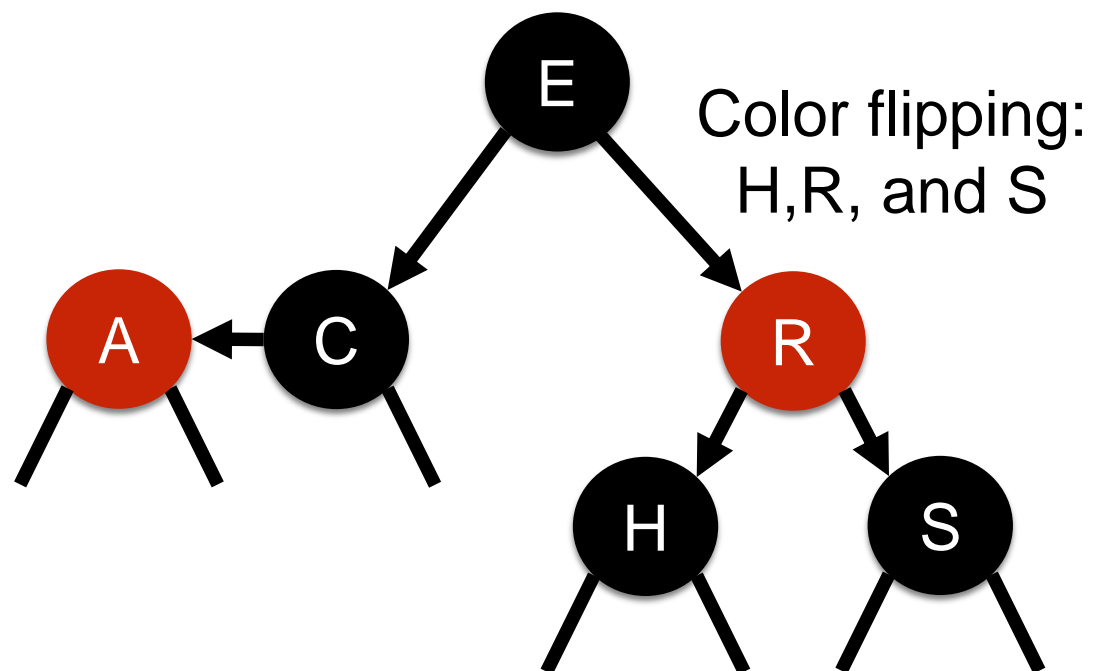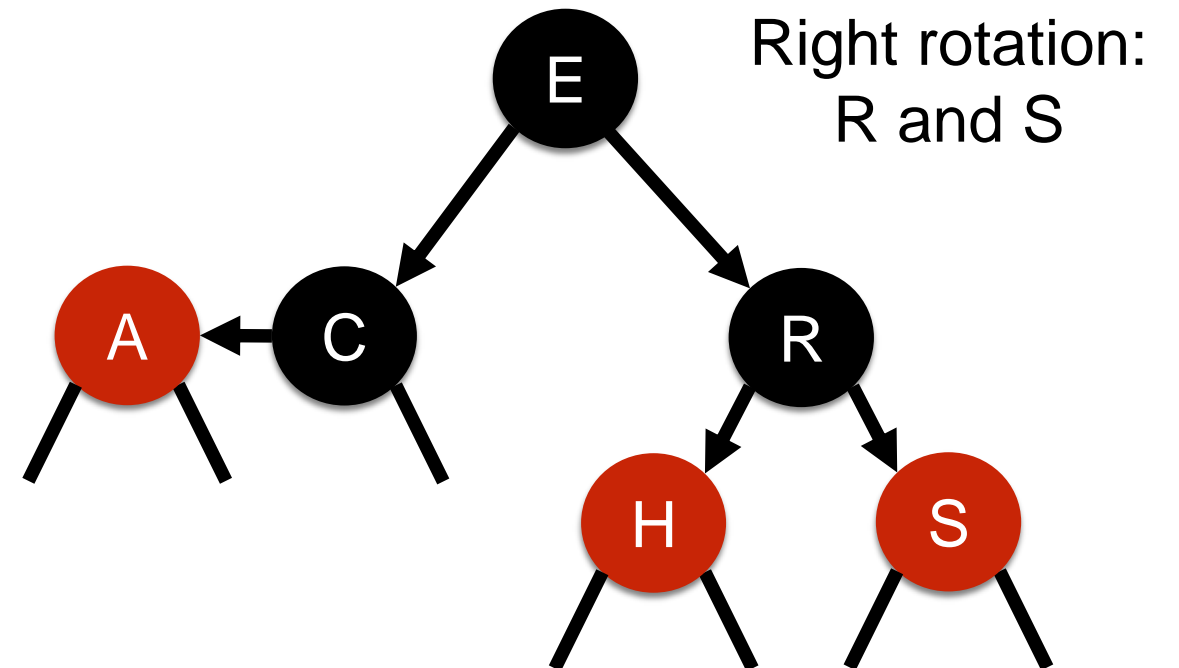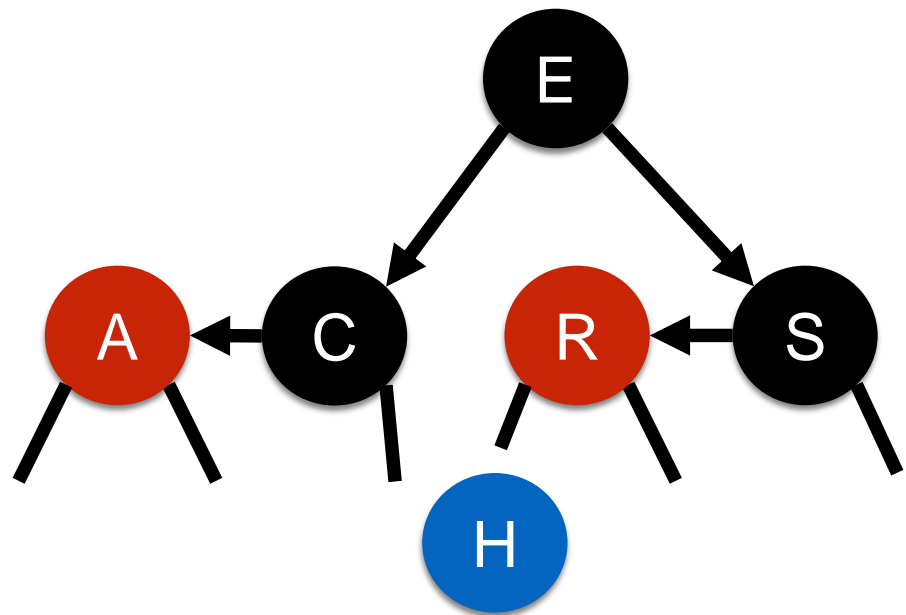
Left rotation: E and R

# Recursion

- When flipping colors, need to fix parent

- If parent is a 3-node, we need to recursively handle the propagation of RED color up to a 2-node or the root

- Handling a RED child in a 3-node is equivalent to handling insertion of new RED node

Right Rotation

Left Rotation

Color flipping

Apply recursively to parent till 2-node, or root

# Deletion

- "Insertion" in a RBT is the *easy* part…

- … the *fun* begins with the "Deletion" operator

- As Insertion is already complicated enough, we will not see Deletion in this course

  - But look at book/code if you are curious

# Homework

- Study Book Chapter 3.1, 3.2 and 3.3

- Study code in the *org.pg4200.les05* package

- Do exercises in *exercises/ex05*

- Extra: do exercises in the book