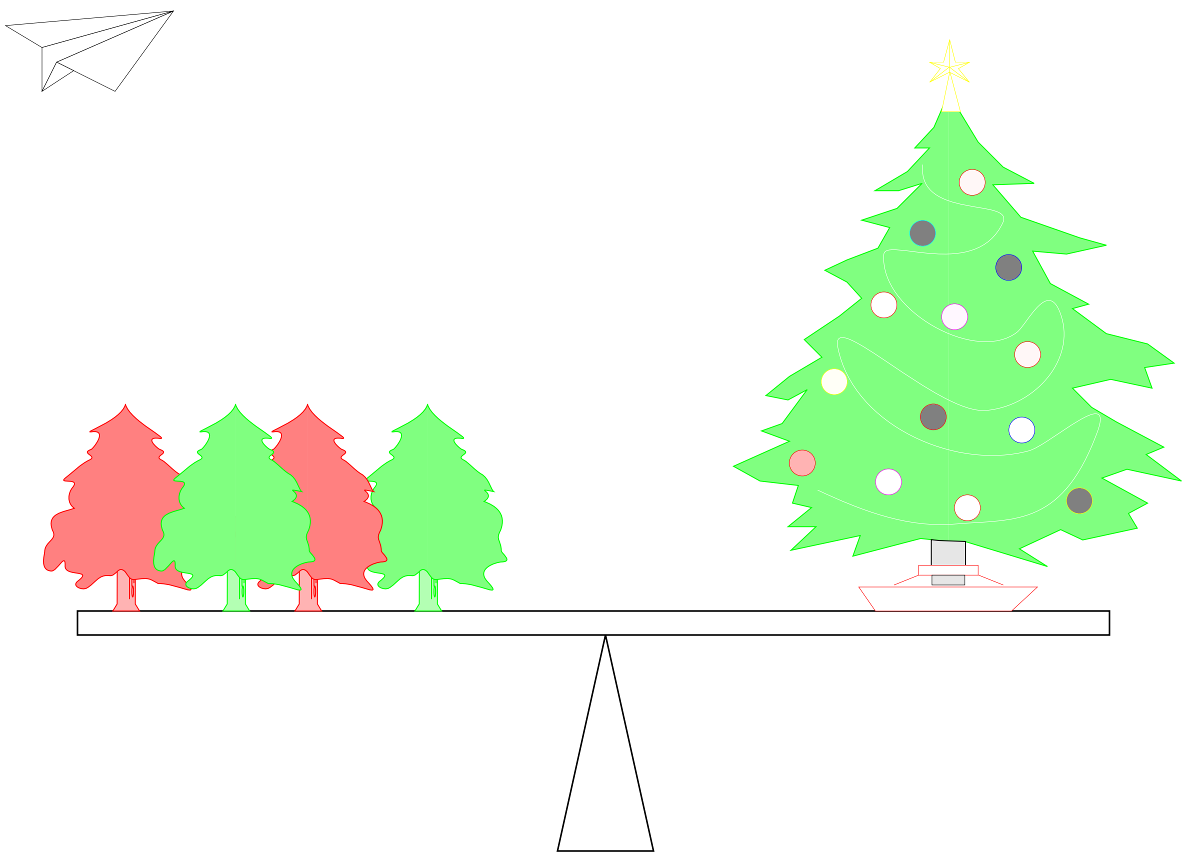
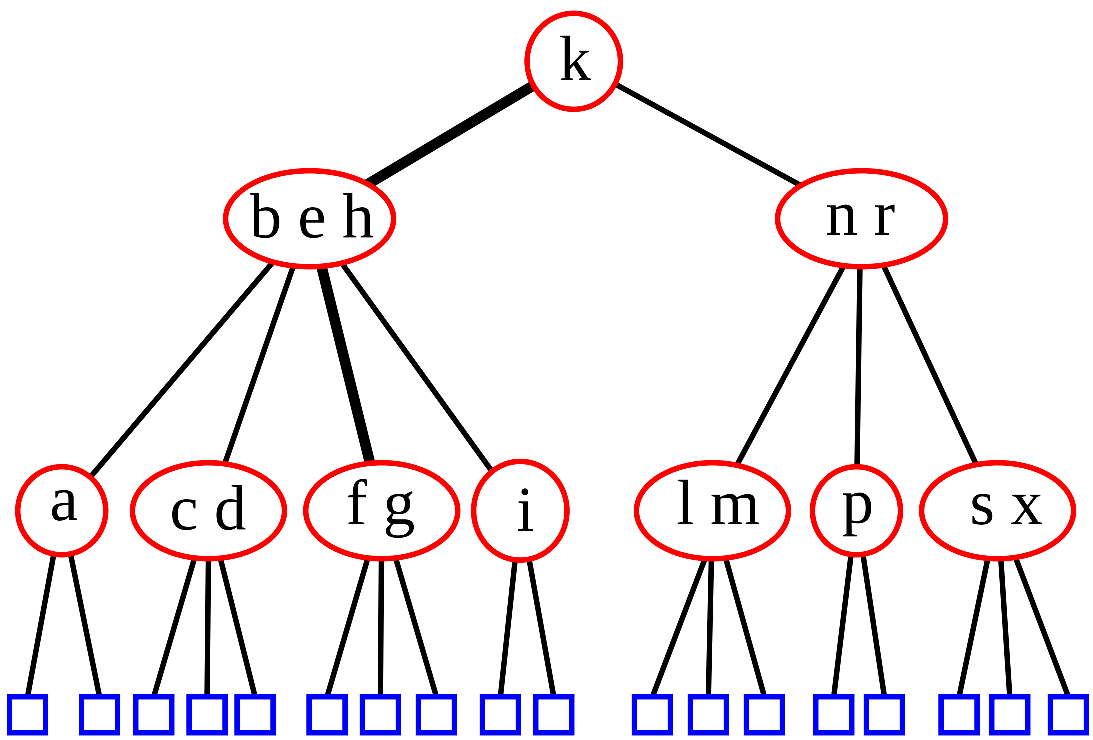
2-3-4 Trees and Red-Black Trees



2-3-4 Trees Revealed

- Nodes store 1, 2, or 3 keys and have 2, 3, or 4 children, respectively
- All leaves have the same depth



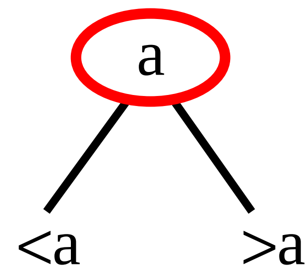
$$\frac{1}{2}\log(N+1) \leq \text{height} \leq \log(N+1)$$

2-3-4 Tree Nodes

- Introduction of nodes with more than 1 key, and more than 2 children

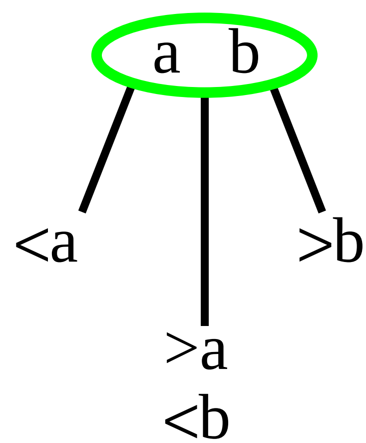
2-node:

- same as a binary node



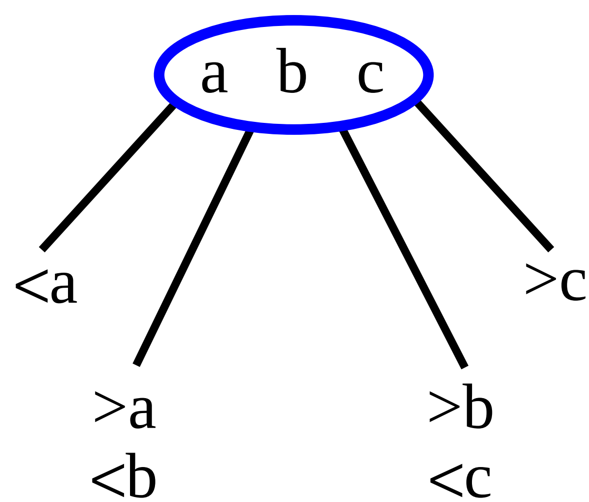
3-node:

- 2 keys, 3 links



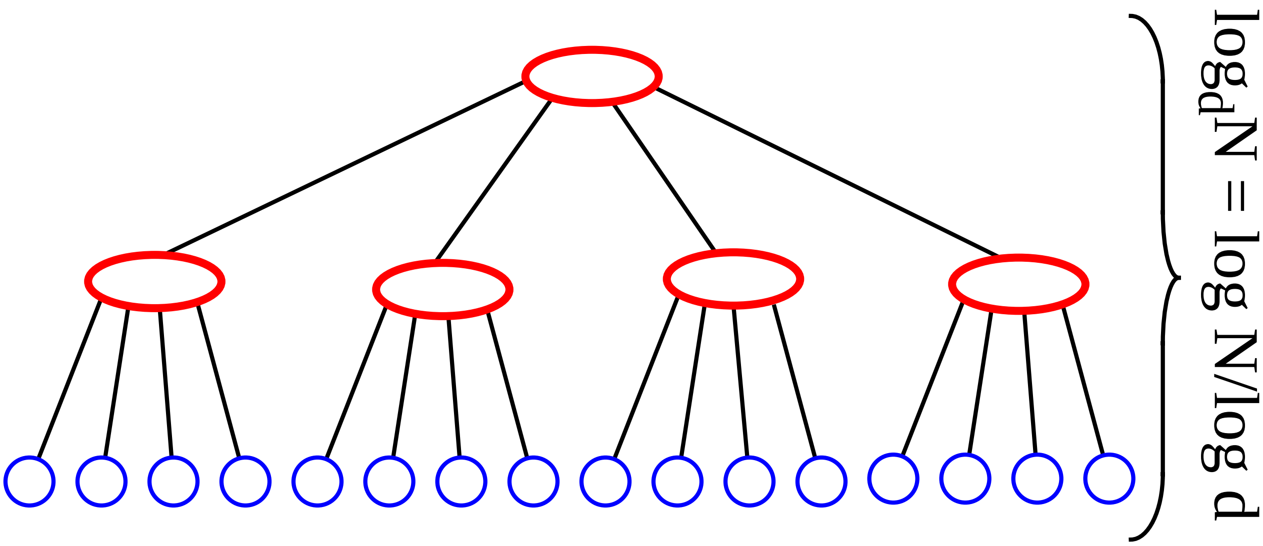
4-node:

- 3 keys, 4 links



Why 2-3-4?

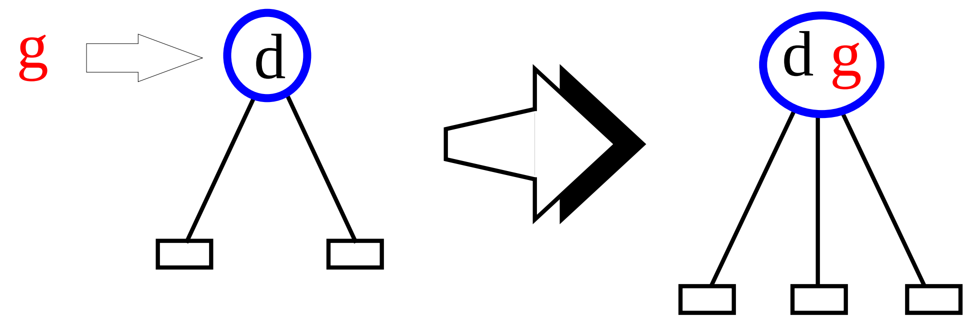
- Why not minimize height by maximizing children in a “d-tree”?
- Let each node have d children so that we get O($\log_d$ N) search time! Right?



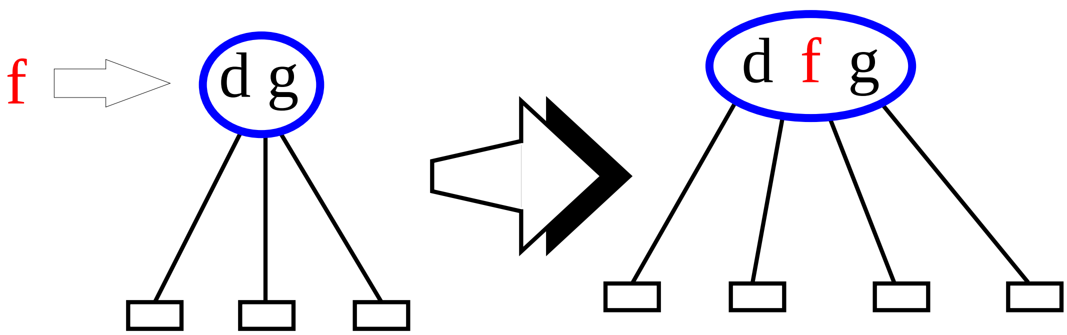
- That means if d = $N^{1/2}$, we get a height of 2
- However, searching out the correct child on each level requires O(log $N^{1/2}$) by binary search
- 2 log $N^{1/2}$ = O(log N) which is not as good as we had hoped for!
- 2-3-4-trees will guarantee O(log N) height using only 2, 3, or 4 children per node

Insertion into 2-3-4 Trees

- Insert the new key at the lowest internal node reached in the search
    - ***2-node*** becomes ***3-node***



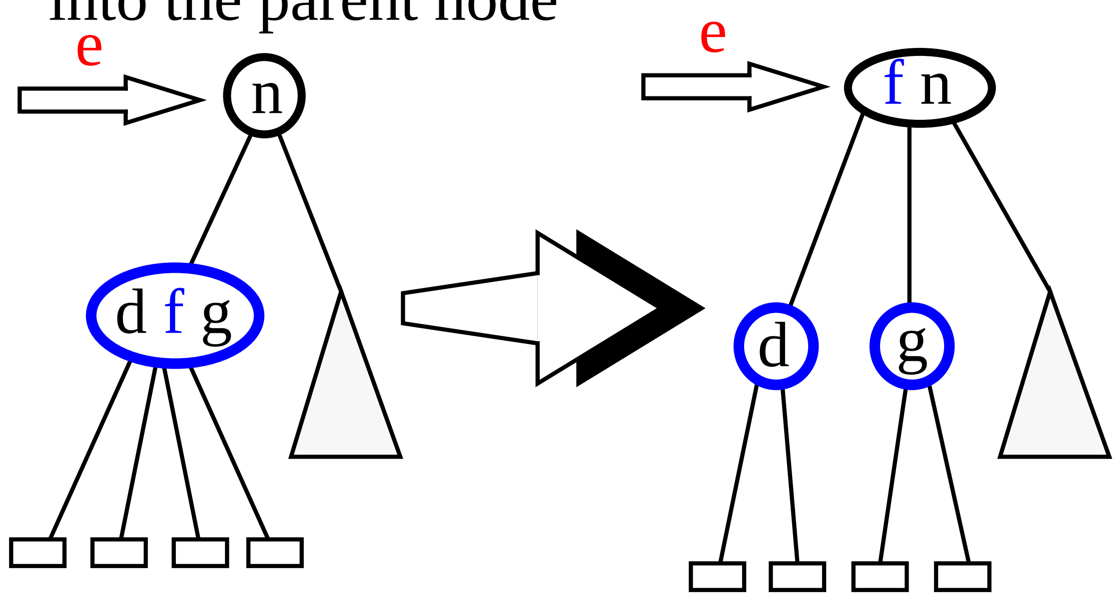
- ***3-node*** becomes ***4-node***

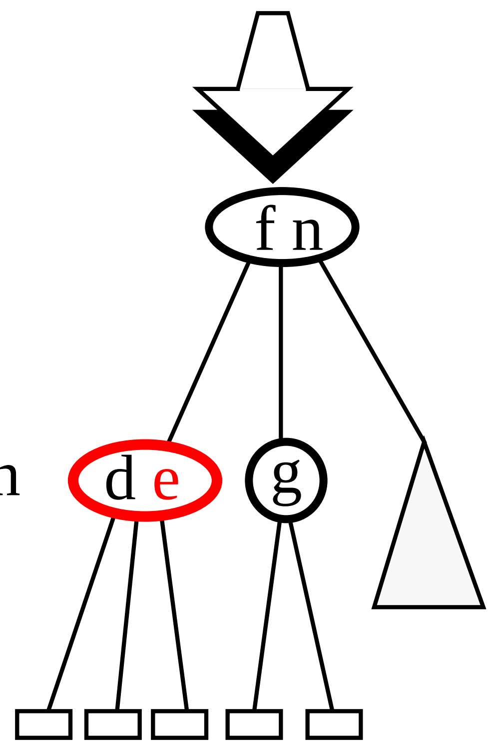


- What about a ***4-node***?
    - We can’t insert another key!

Top Down Insertion

- In our way down the tree, whenever we reach a ***4-node***, we break it up into two ***2-nodes***, and move the middle element up into the parent node

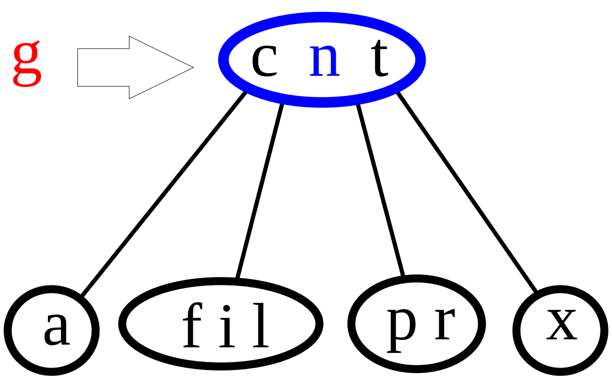


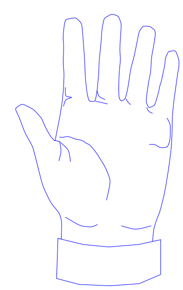


- Now we can perform the insertion using one of the previous two cases
- Since we follow this method from the root down to the leaf, it is called ***top down insertion***

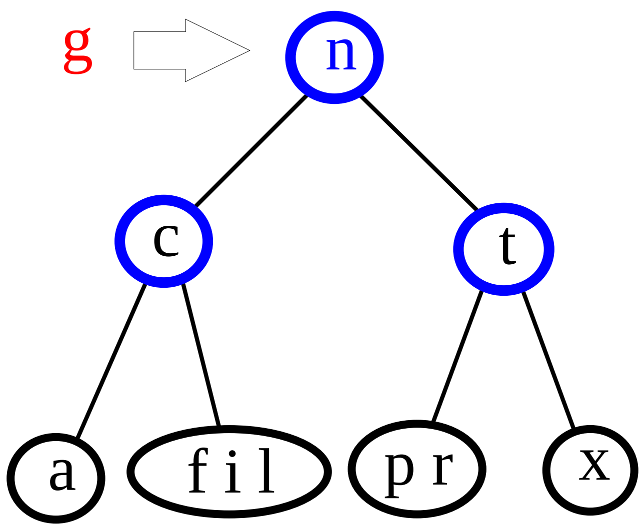
Splitting the Tree

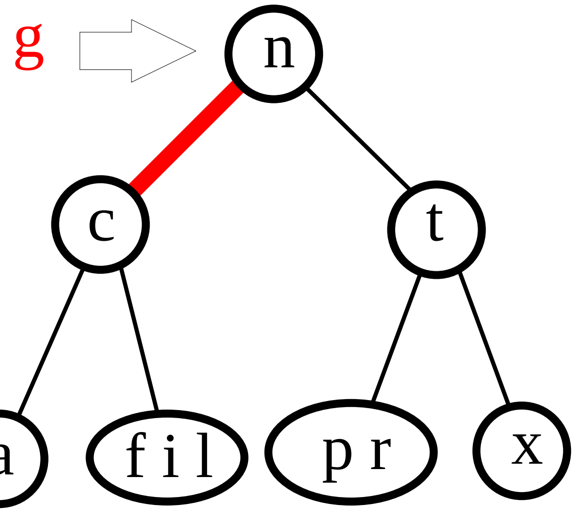
As we travel down the tree, if we encounter any ***4-node*** we will break it up into ***2-nodes***. This guarantees that we will never have the problem of inserting the middle element of a former ***4-node*** into its parent ***4-node***.

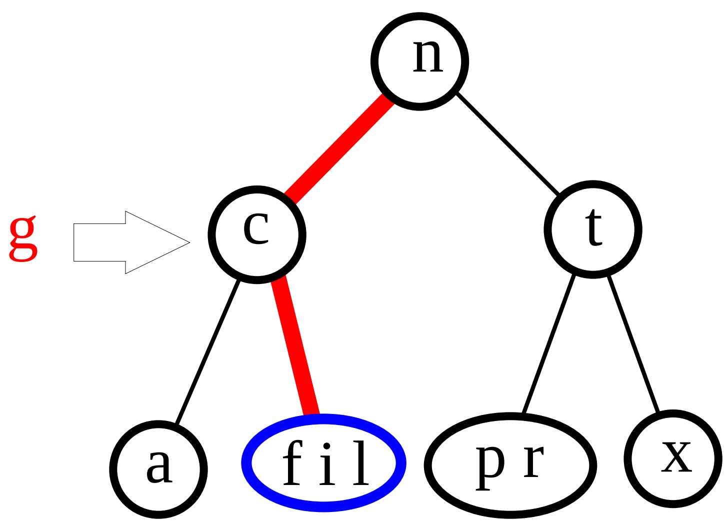


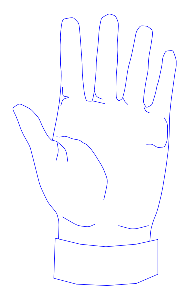


Whoa, cowboy

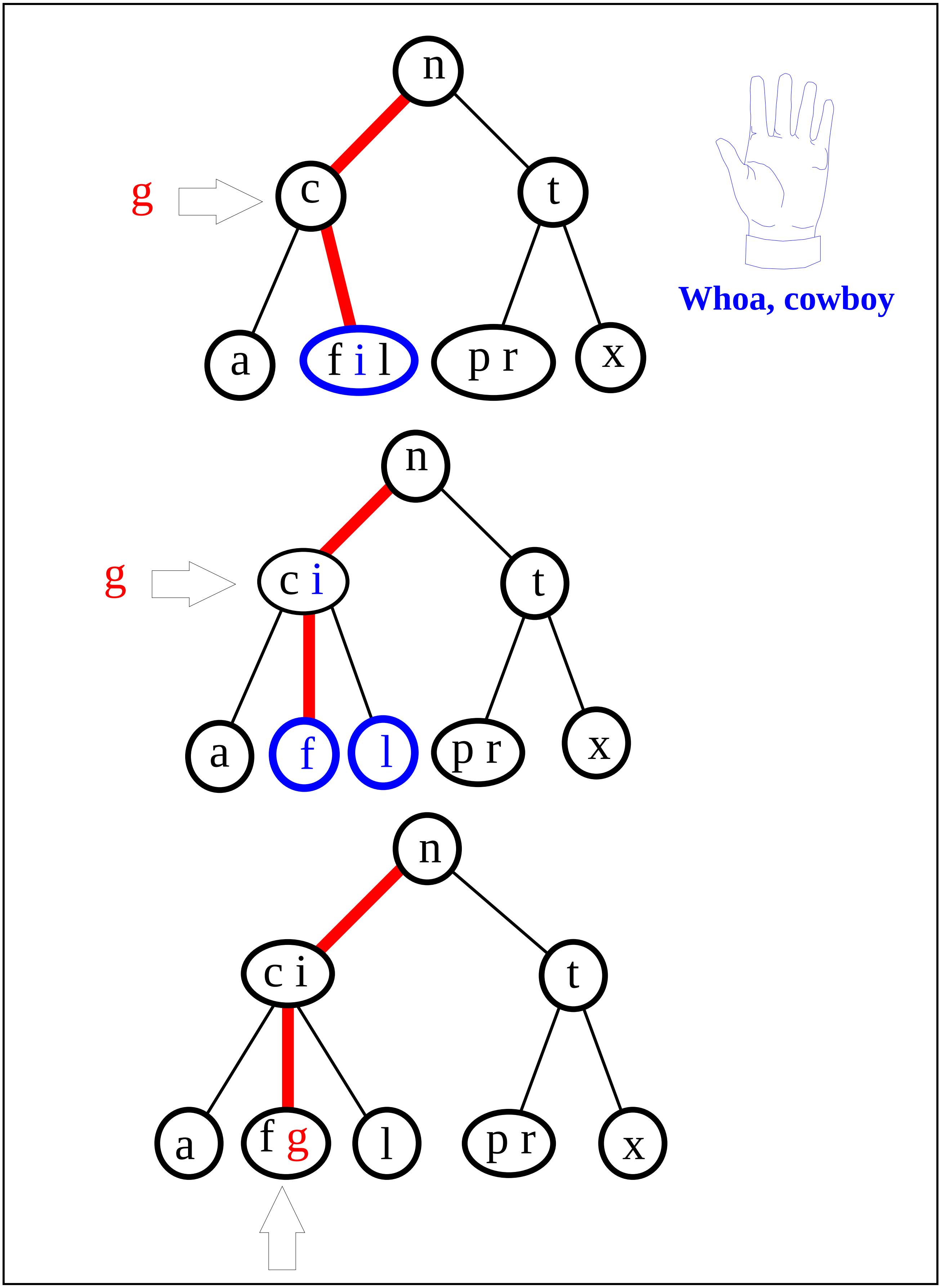








Whoa, cowboy


n
g
c
t
a
f i l
p r
x
Whoa, cowboy
n
g
c i
t
a
f
l
p r
x
n
c i
t
a
f g
l
p r
x

Time Complexity of Insertion in 2-3-4 Trees

Time complexity:

- A search visits O(log N) nodes
- An insertion requires O(log N) node splits
- Each node split takes constant time
- Hence, operations ***Search*** and ***Insert*** each take time O(log N)

Notes:

- Instead of doing splits top-down, we can perform them bottom-up starting at the insertion node, and only when needed. This is called ***bottom-up*** insertion.
- A deletion can be performed by ***fusing*** nodes (inverse of splitting), and takes O(log N) time

Beyond 2-3-4 Trees

What do we know about 2-3-4 Trees?

- Balanced

- O(log N) search time

- Different node structures

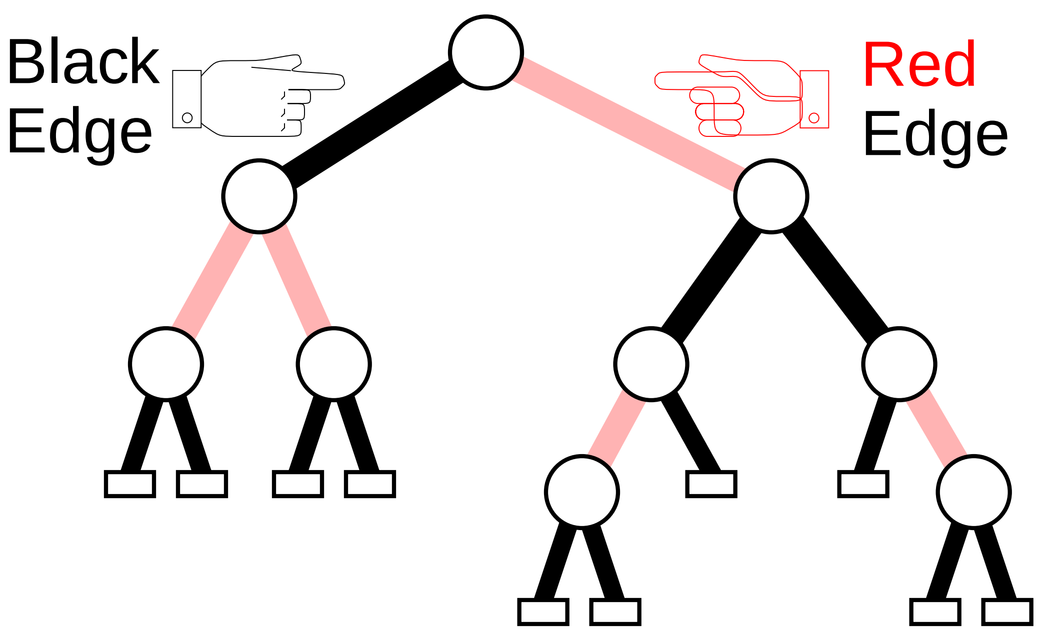
Can we get 2-3-4 tree advantages in a binary tree format???

Welcome to the world of Red-Black Trees!!!

Red-Black Tree

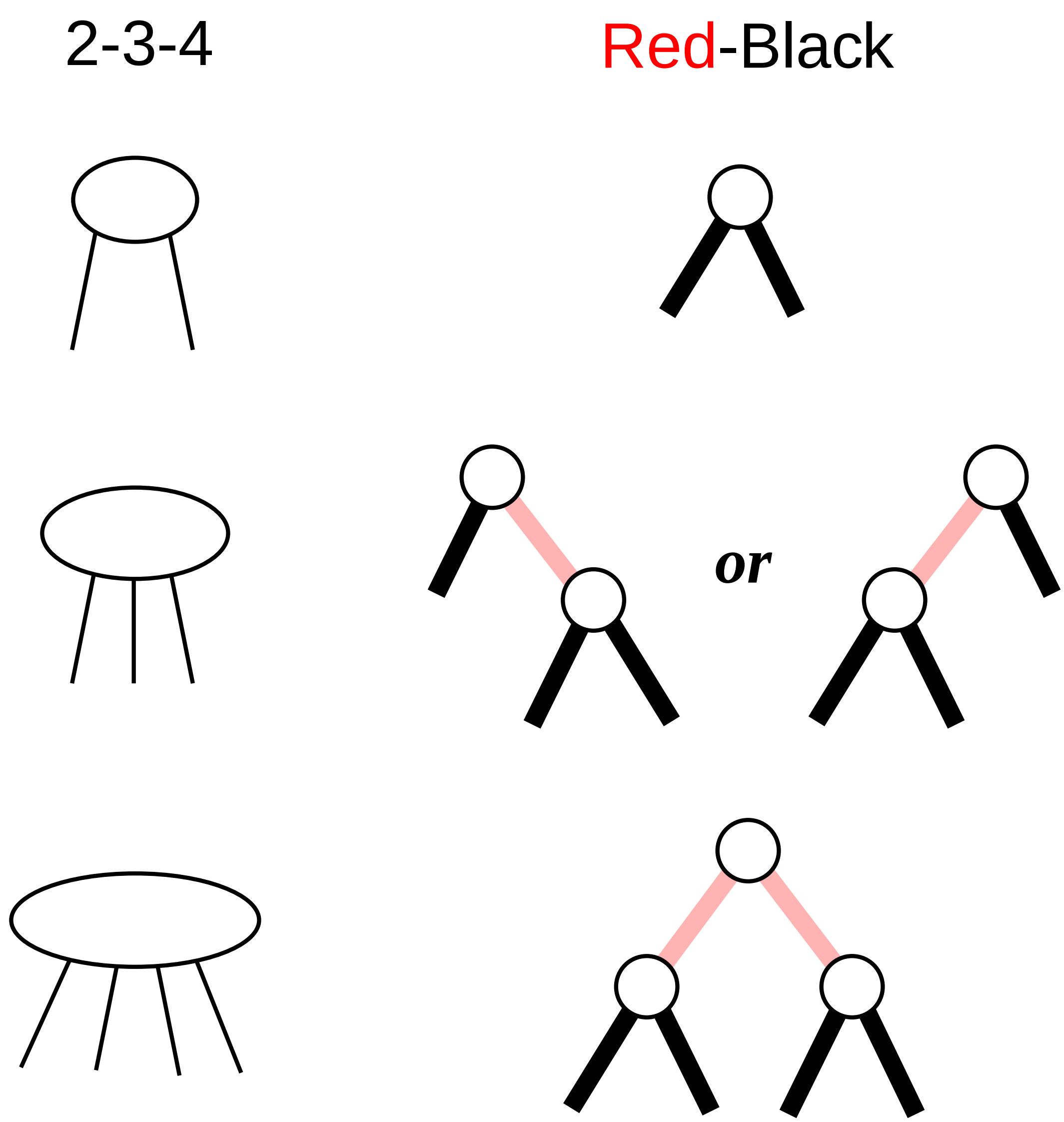
A **red-black tree** is a binary search tree with the following properties:

- edges are colored **red** or **black**
- ***no two consecutive red edges*** on any root-leaf path
- ***same number of black edges*** on any root-leaf path (= ***black height*** of the tree)
- ***edges connecting leaves are black***



2-3-4 Tree Evolution

Note how **2-3-4 trees** relate to **red-black trees**

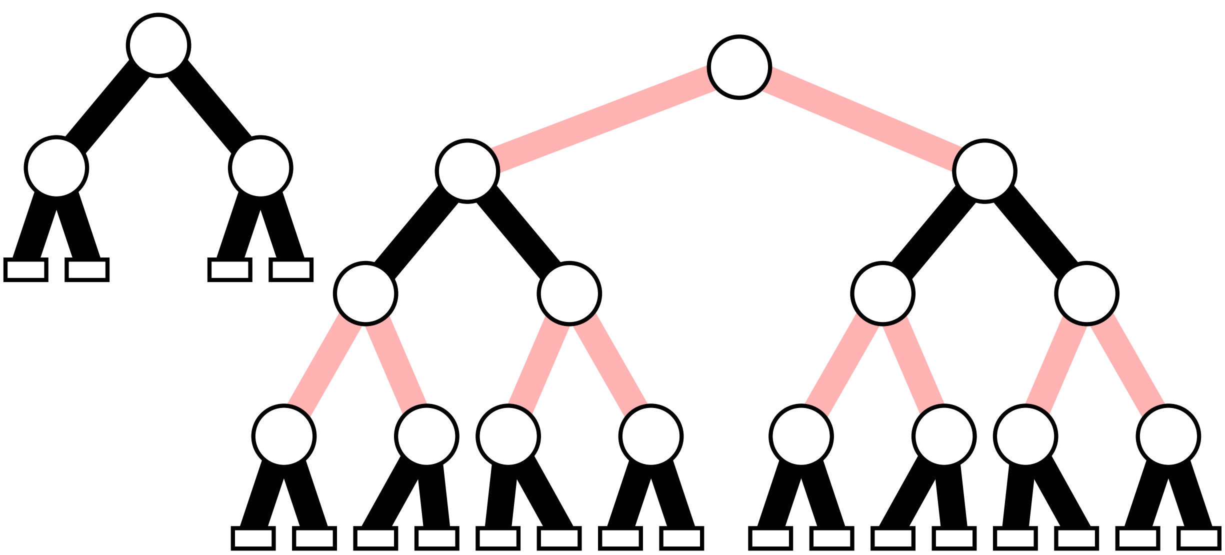


Now we see red-black trees are just a way of representing 2-3-4 trees!

More Red-Black Tree Properties

N   # of internal nodes
L   # leaves (= N + 1)
H   height
B   black height

Property 1: $2^B \leq N + 1 \leq 4^B$



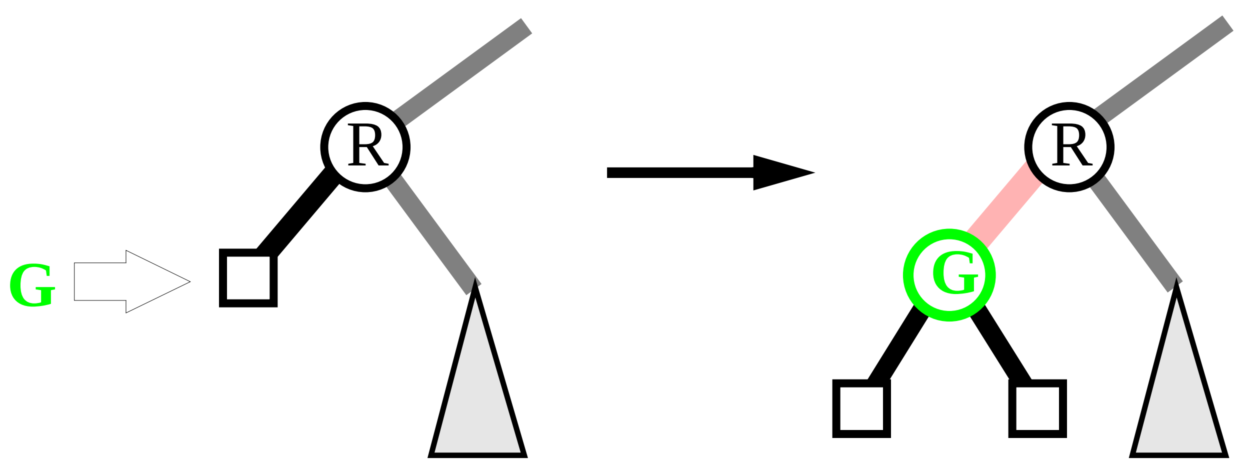
Property 2: $\frac{1}{2}\log(N+1) \leq B \leq \log(N+1)$

Property 3: $\log(N+1) \leq H \leq 2\log(N+1)$

This implies that searches take time O(logN)!

Insertion into Red-Black Trees

1. Perform a standard search to find the leaf where the key should be added
2. Replace the leaf with an internal node with the new key
3. Color the incoming edge of the new node red
4. Add two new leaves, and color their incoming edges black
5. If the parent had an incoming red edge, we now have two consecutive red edges! We must reorganize tree to remove that violation. What must be done depends on the sibling of the parent.

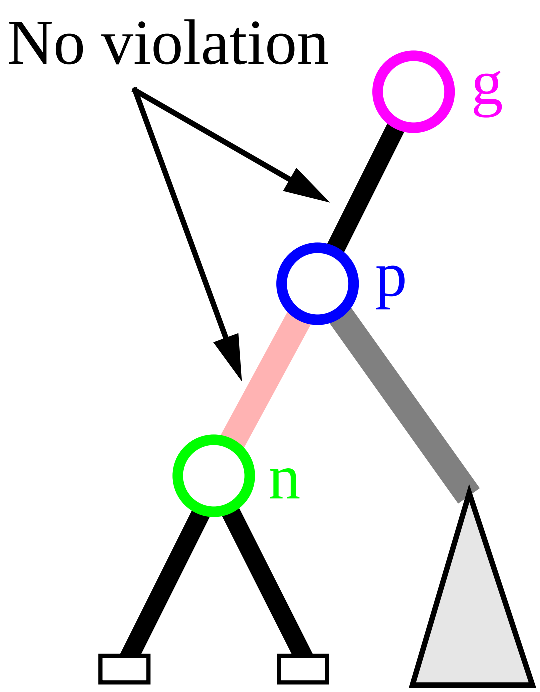


Insertion - Plain and Simple

Let:

- n be the new node
- p be its parent
- g be its grandparent

Case 1: Incoming edge of p is black



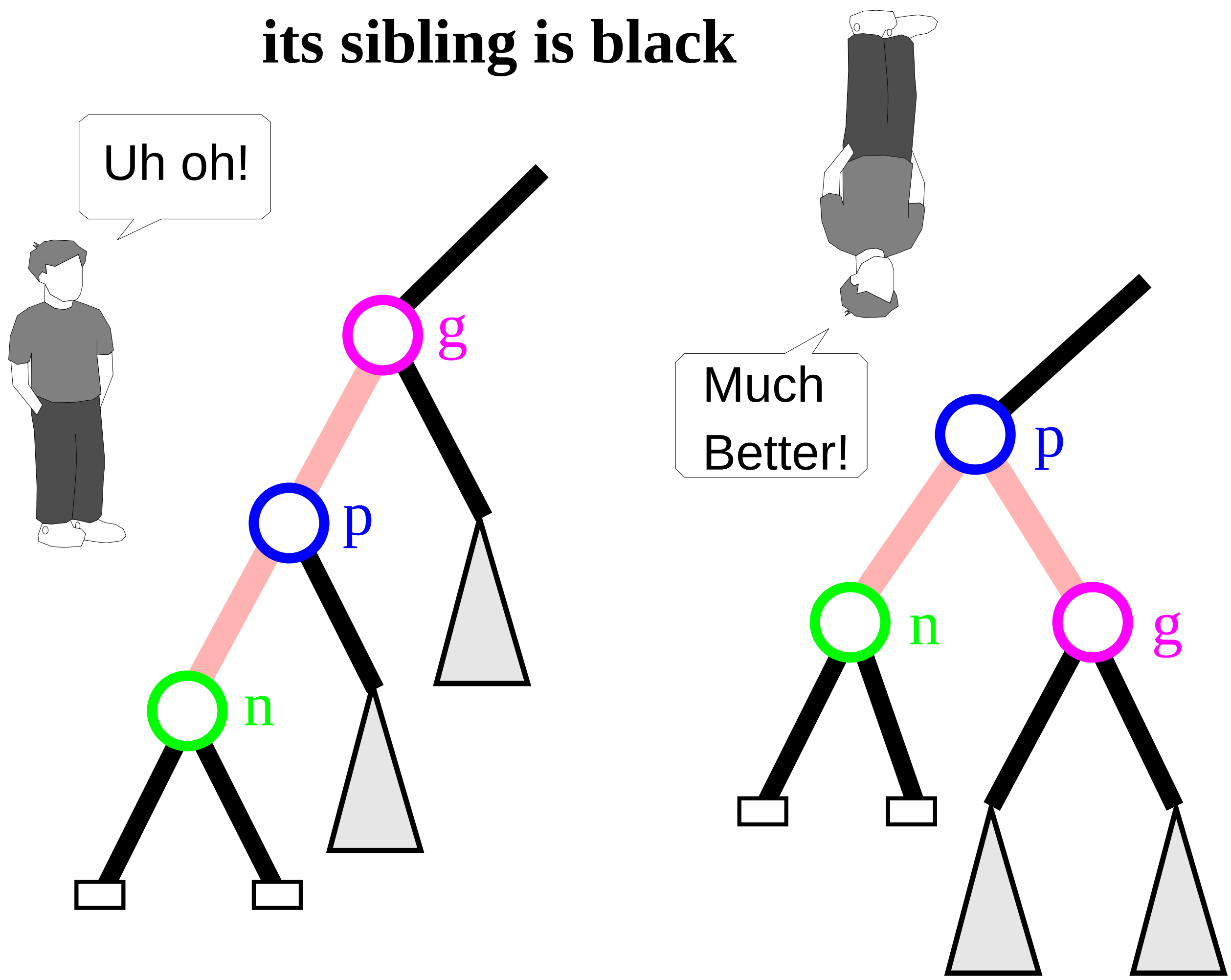
STOP!

Pretty easy, huh?

Well... it gets messier...

Restructuring

Case 2: Incoming edge of p is red, and its sibling is black

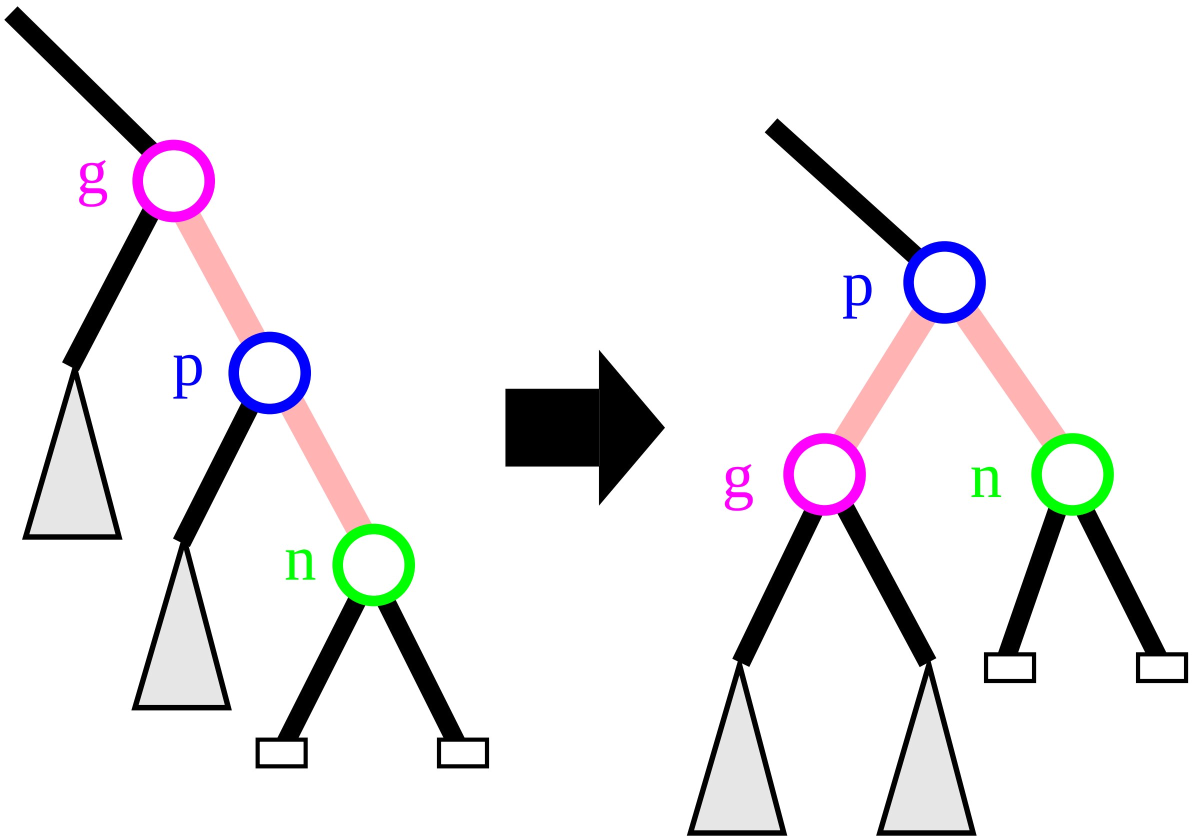


We call this a "***right rotation***"

- No further work on tree is necessary
- Inorder remains unchanged
- Tree becomes more balanced
- No two consecutive red edges!

More Rotations

Similar to a right rotation, we can do a
"**left rotation**"...



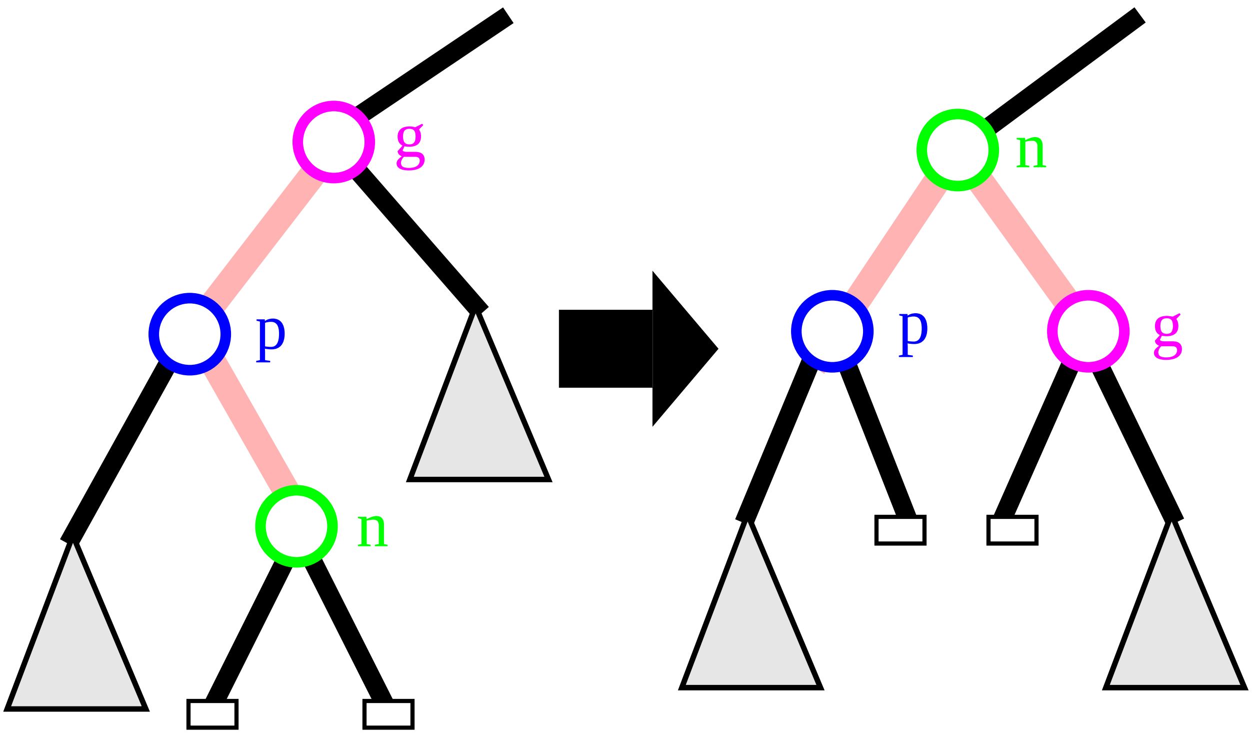
Simple, huh?

Double Rotations

What if the new node is between its parent and grandparent in the inorder sequence?

We must perform a "double rotation"

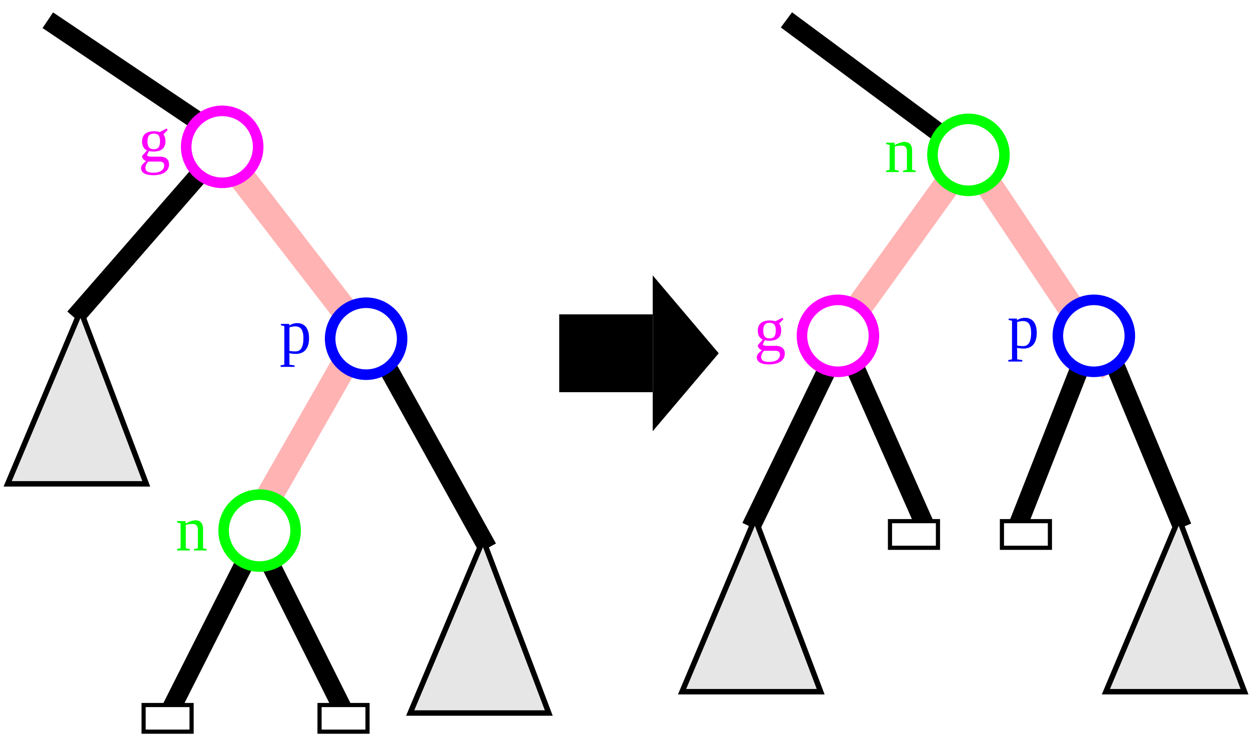
(which is no more difficult than a "single" one)



This would be called a
"left-right double rotation"

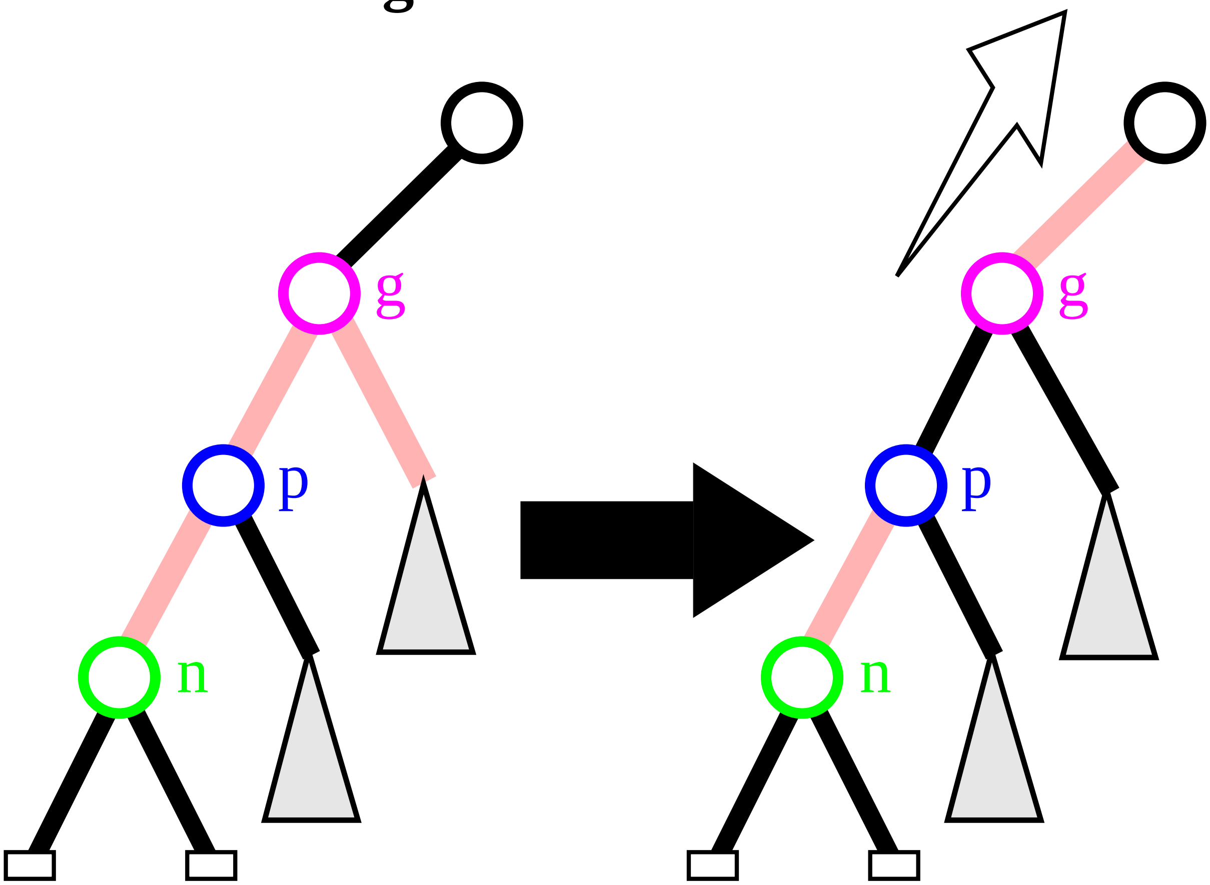
Last of the Rotations

And this would be called a
"right-left double rotation"



Bottom-Up Rebalancing

Case 3: Incoming edge of p is red and its sibling is also red



- We call this a "***promotion***"
- Note how the black depth remains unchanged for all of the descendants of **g**
- This process will continue upward beyond **g** if necessary: rename **g** as **n** and repeat.

Summary of Insertion

- If two red edges are present, we do either
  - a ***restructuring*** (with a simple or double rotation) and **stop**, or
  - a ***promotion*** and **continue**

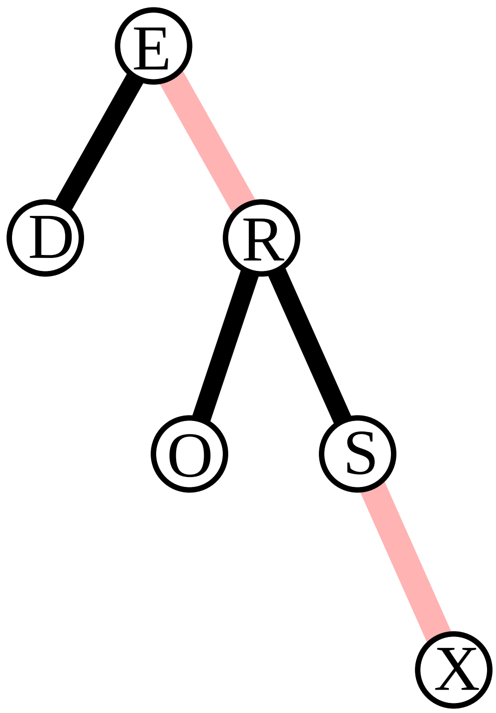
- A restructuring takes constant time and is performed at most once. It reorganizes an off-balanced section of the tree.

- Promotions may continue up the tree and are executed O(log N) times.

- The **time complexity** of an insertion is **O(logN).**

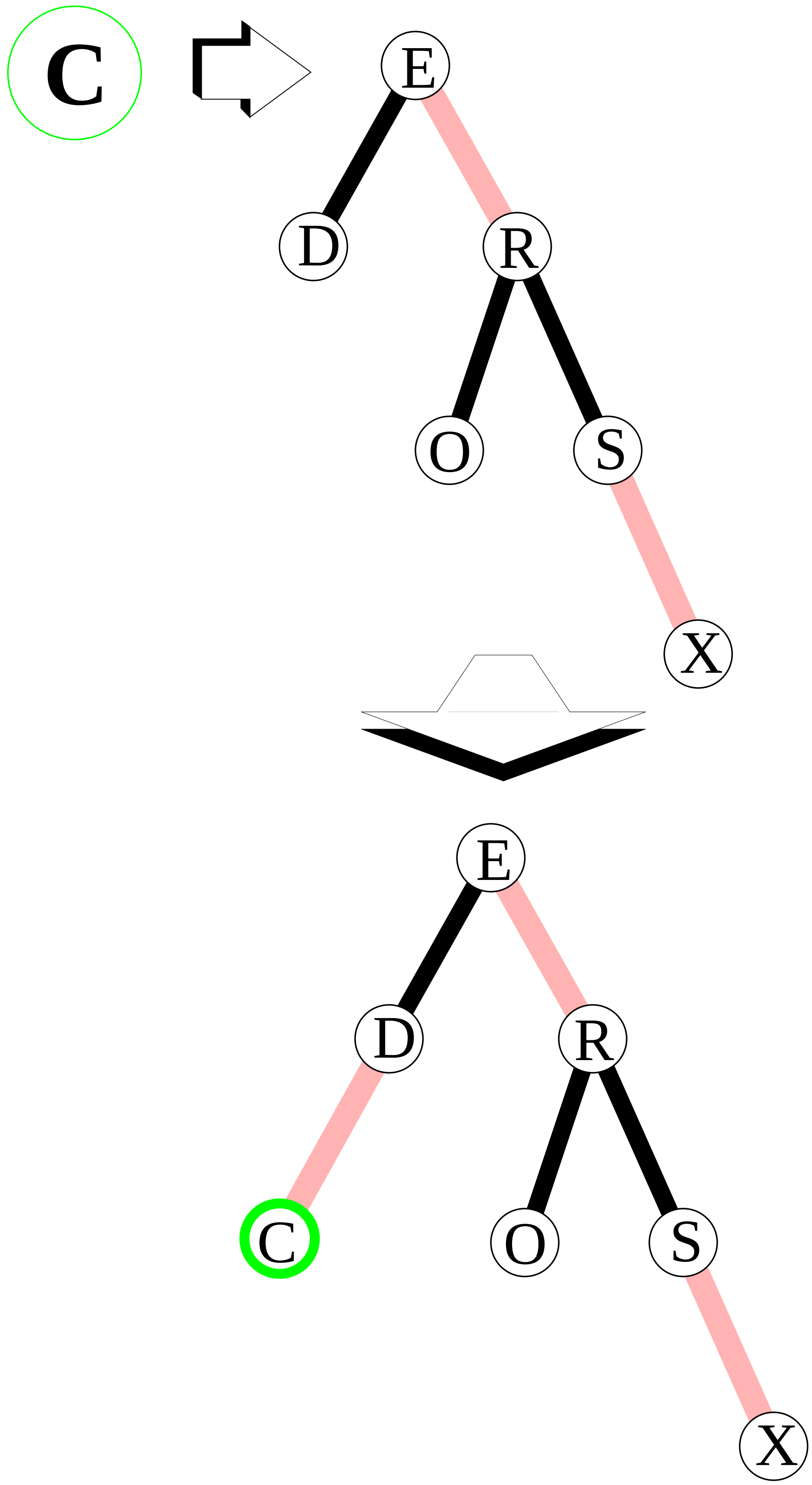
An Example

Start by inserting “REDSOX” into an empty tree

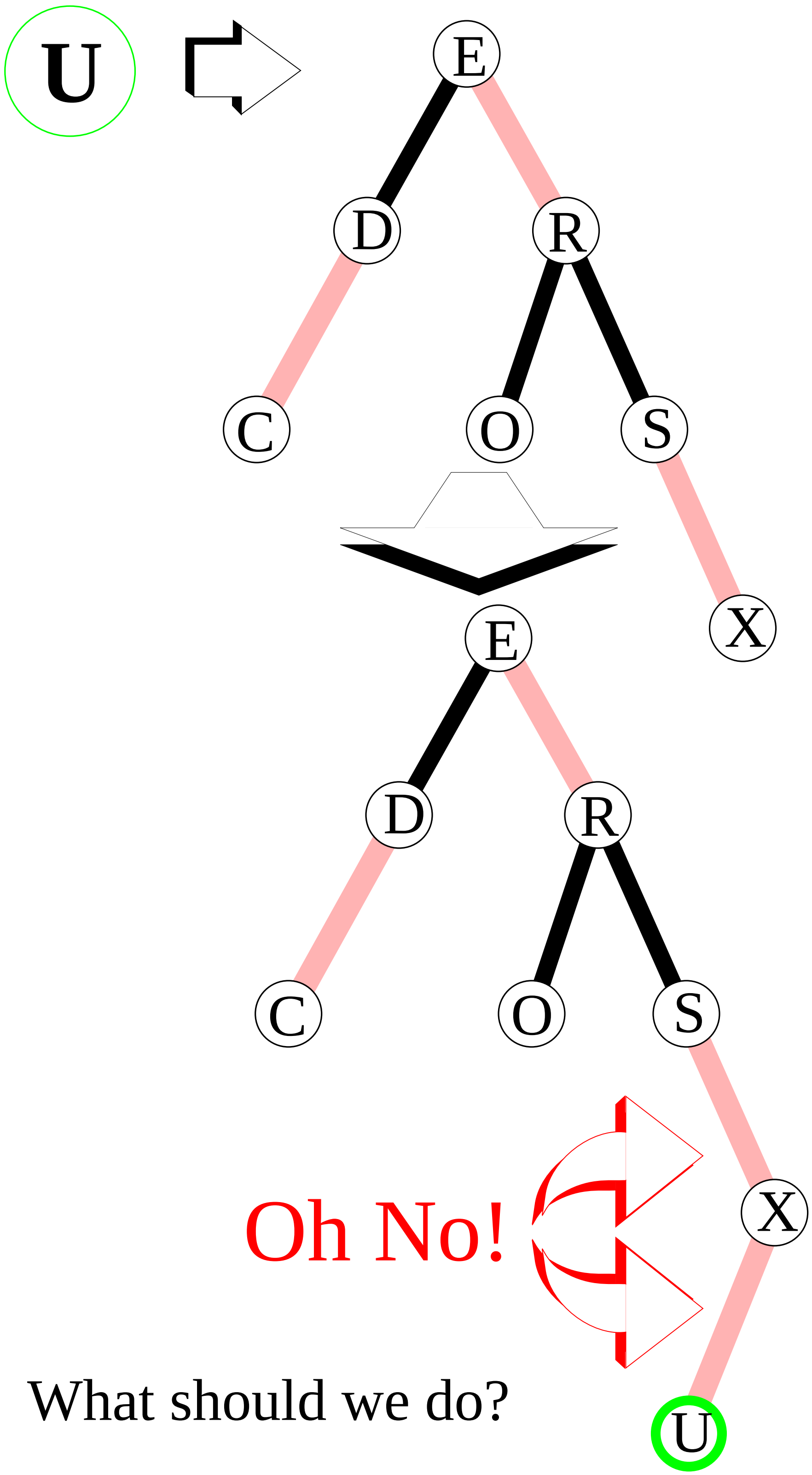


Now, let’s insert “C U B S”...

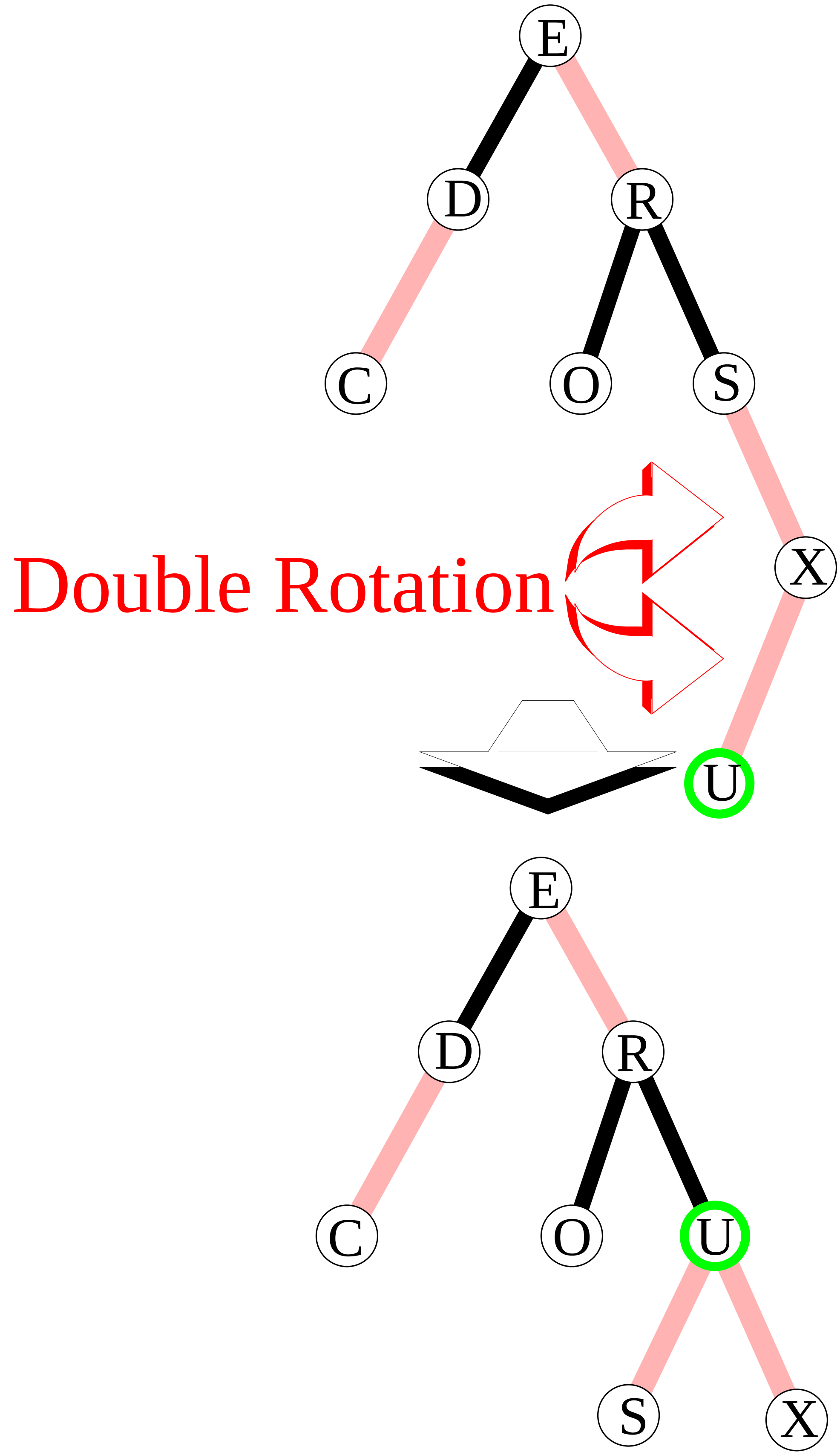
A Cool Example



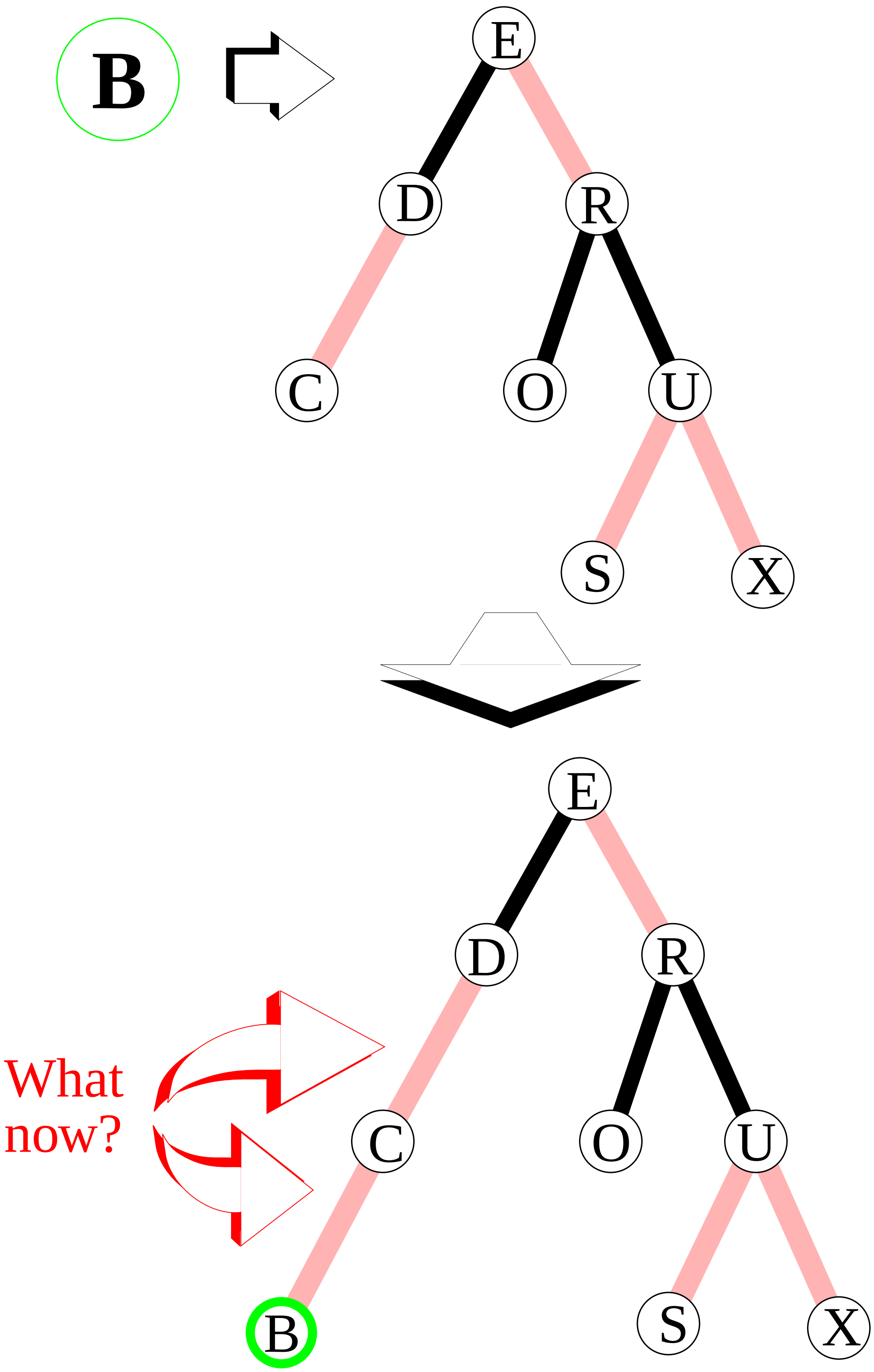
An Unbelievable Example

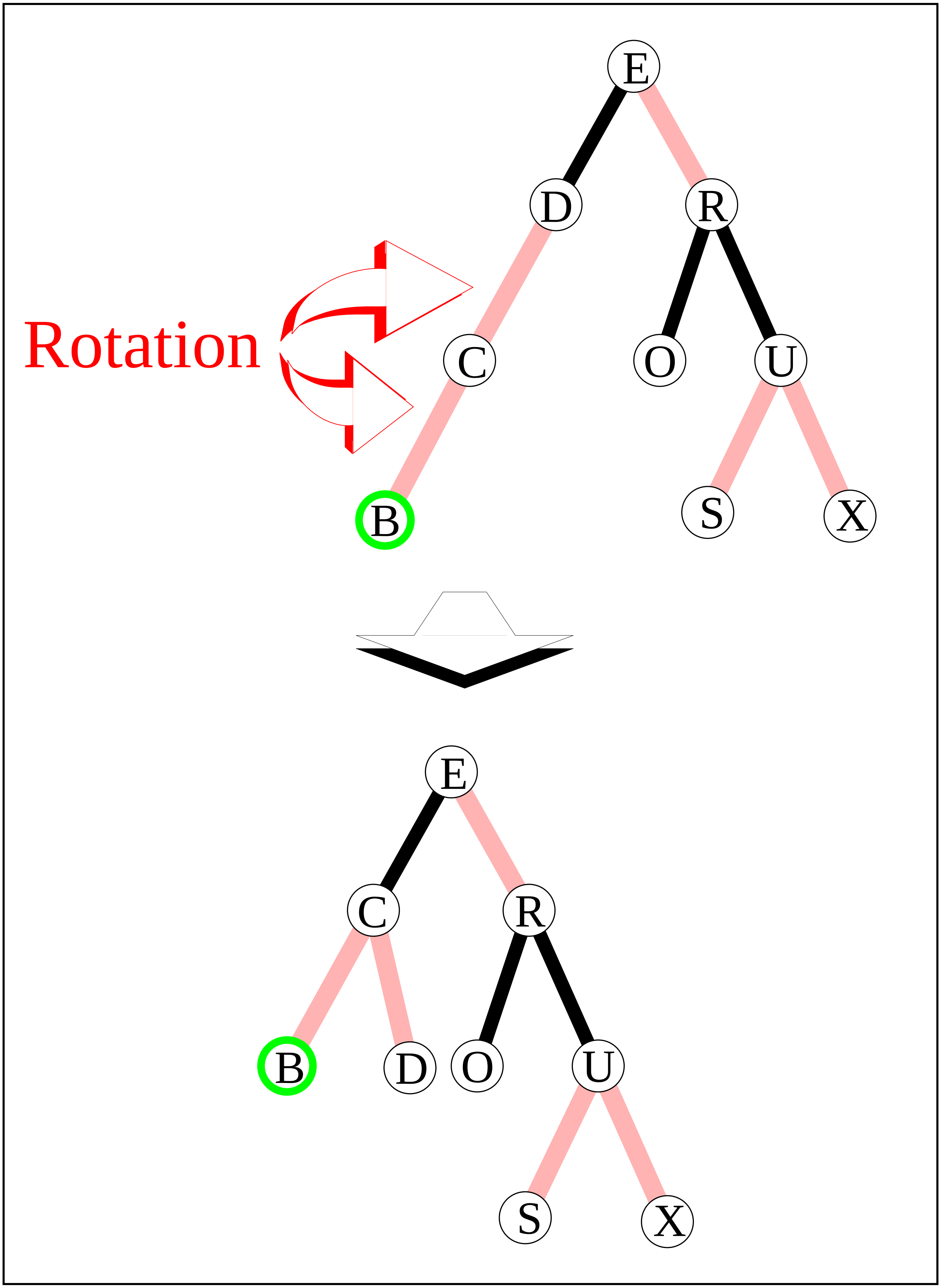


What should we do?


E
D
R
C
O
S
Double Rotation
X
U
E
D
R
C
O
U
S
X

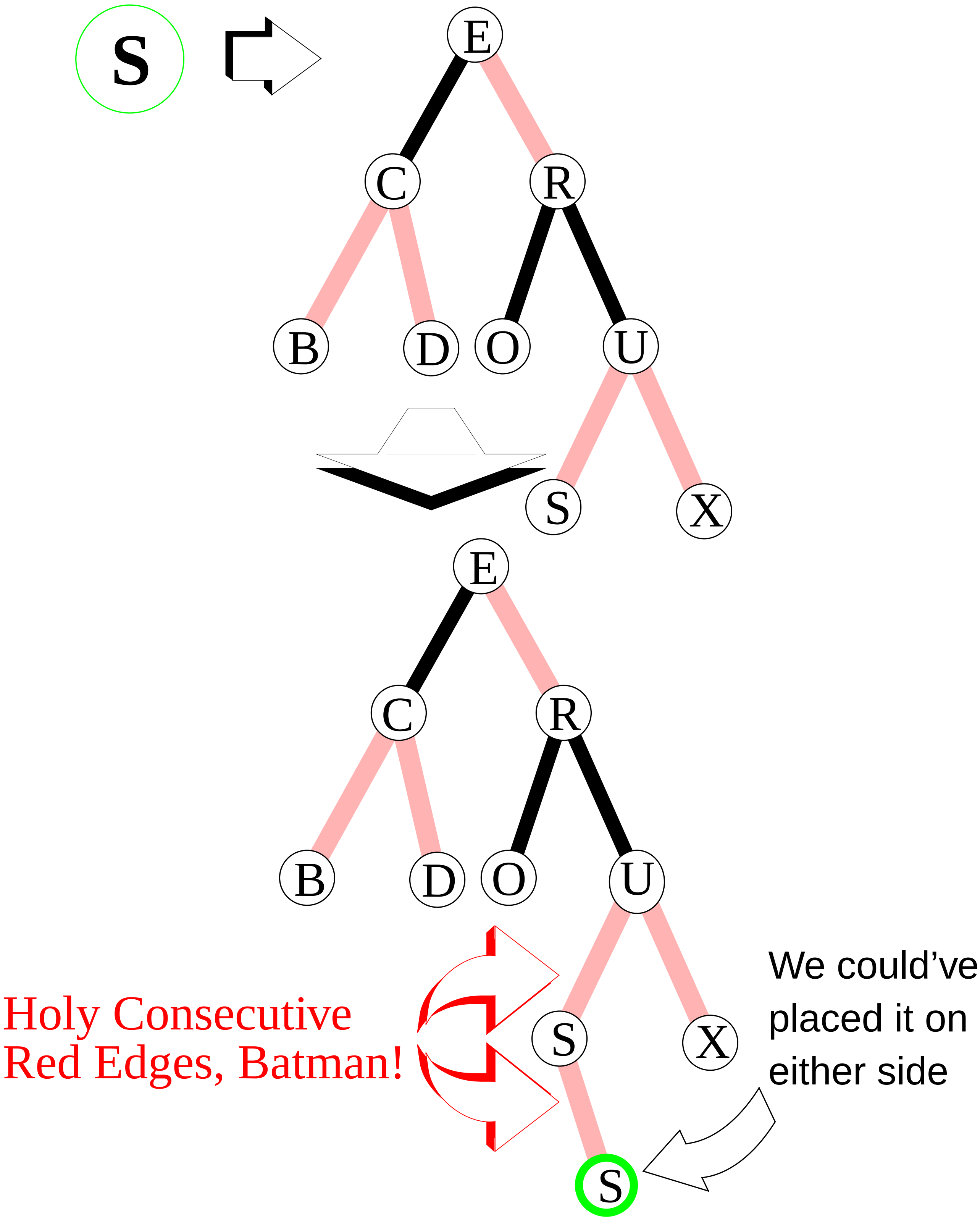
A Beautiful Example

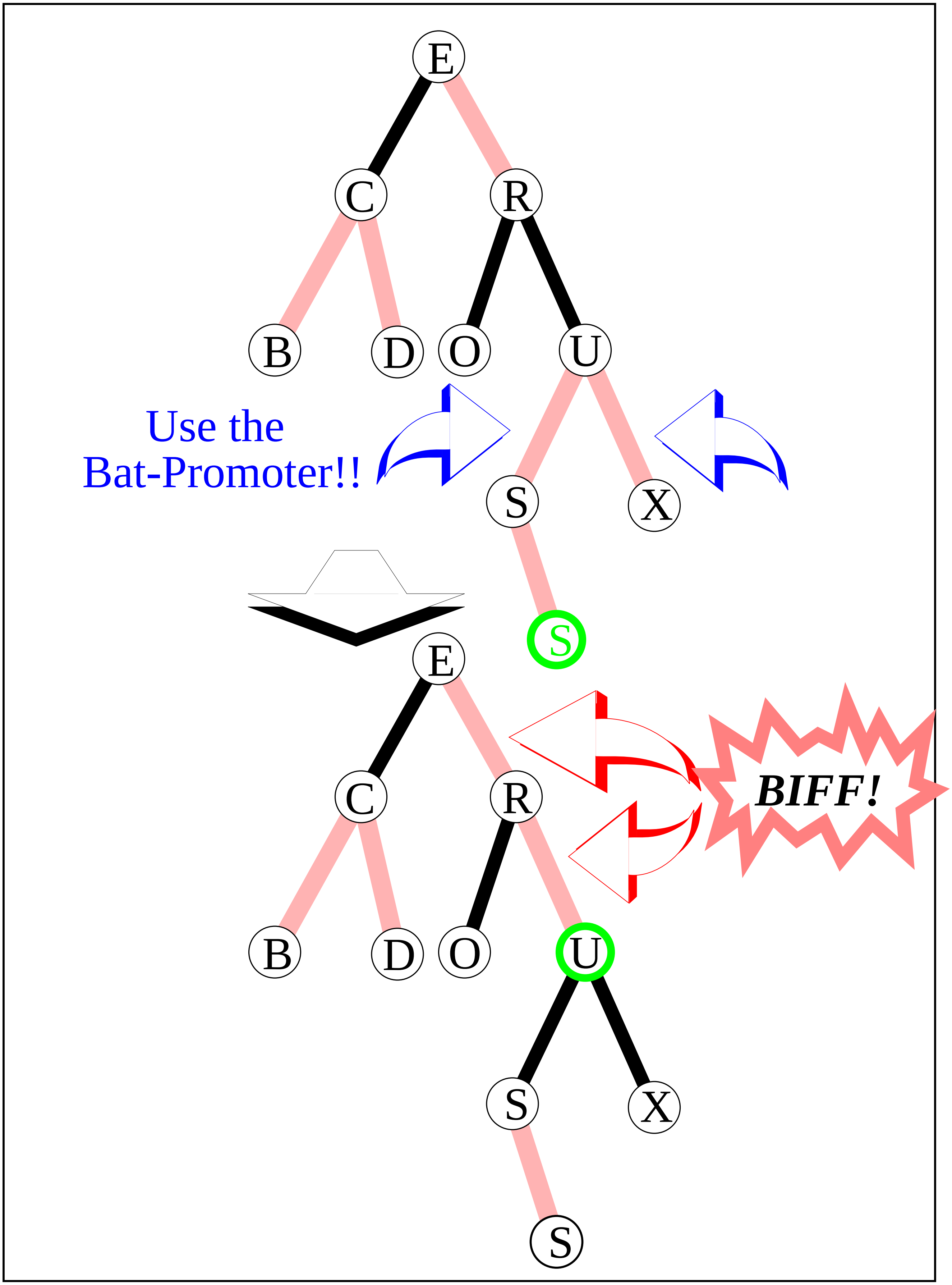


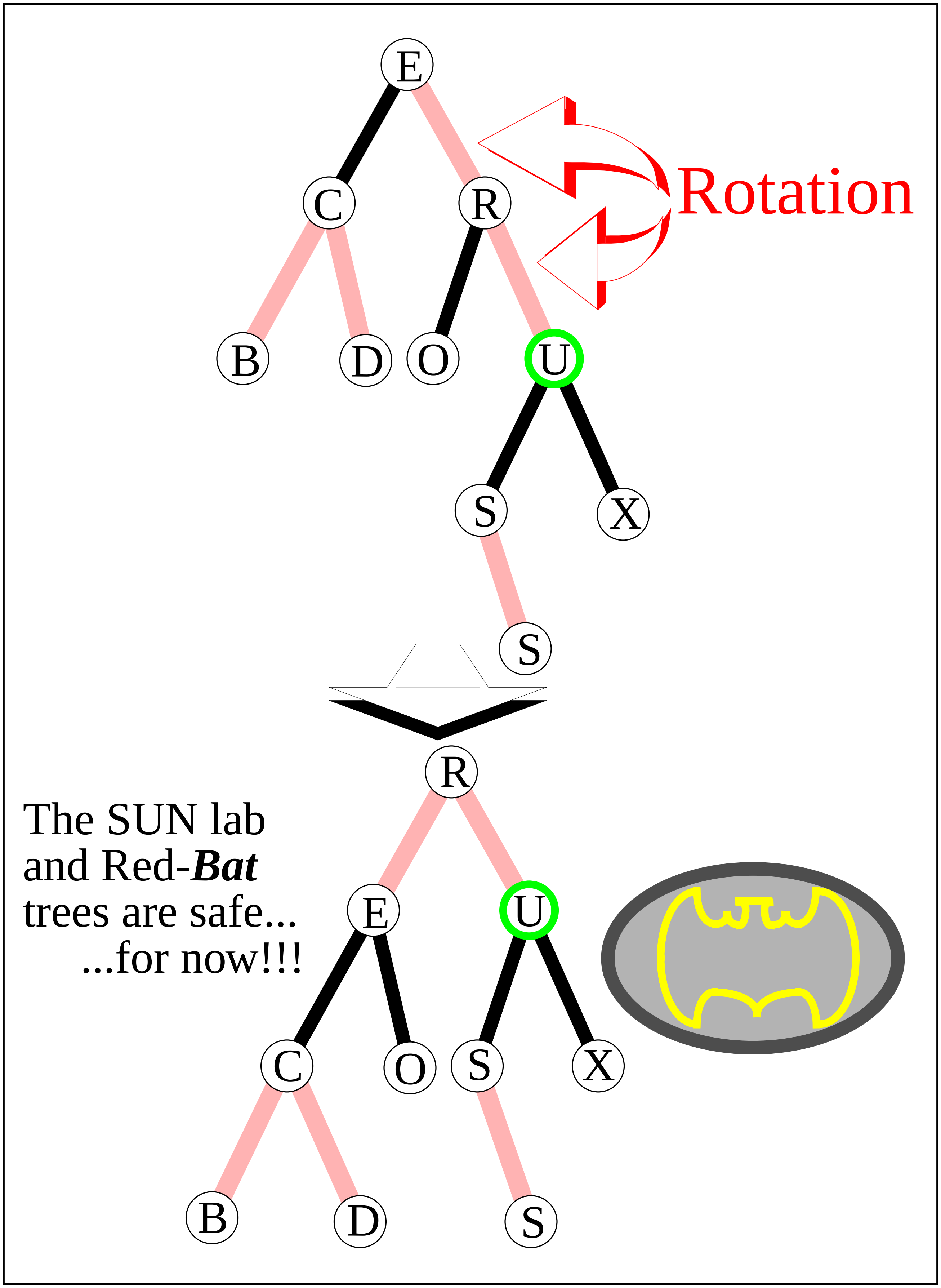


Rotation
E
D
R
C
O
U
B
S
X
E
C
R
B
D
O
U
S
X

A Super Example




E
C
R
B
D
O
U
Use the
Bat-Promoter!!
S
X
S
E
C
R
BIFF!
B
D
O
U
S
X
S


Rotation
The SUN lab
and Red-*Bat*
trees are safe...
...for now!!!