Pre-Announcements

Maggie works at School of Optometry at Berkeley.

- Running Ocular studies and need lots of participants.
- Make $20 to $30 hours an hour. Fliers up front.
-

Announcements

Reminder: Big O vs. Big Theta

- Big O is not the same as worst case.
- Big Omega is not the same as best case.

Midterm 2:

- 3/20, 8-10 PM in various rooms.
- Covers material through 3/16 (next Friday).
- Study using study guides.
  - THE KEY IS METACOGNITION: Reflect on your problem solving strategies and those of your fellow students.
  - Understanding a handful of solutions to old midterm problems is less helpful than you might think -- look at answers as late as possible.
- There is an alternate 61C midterm from 6 - 8 in 1 LeConte.

CS61B

Lecture 22: Balanced Search Trees

- 2-3-4 and 2-3 Trees (a.k.a. B-Trees)
- Tree Rotation
- Red-Black Trees

Daniel Ross

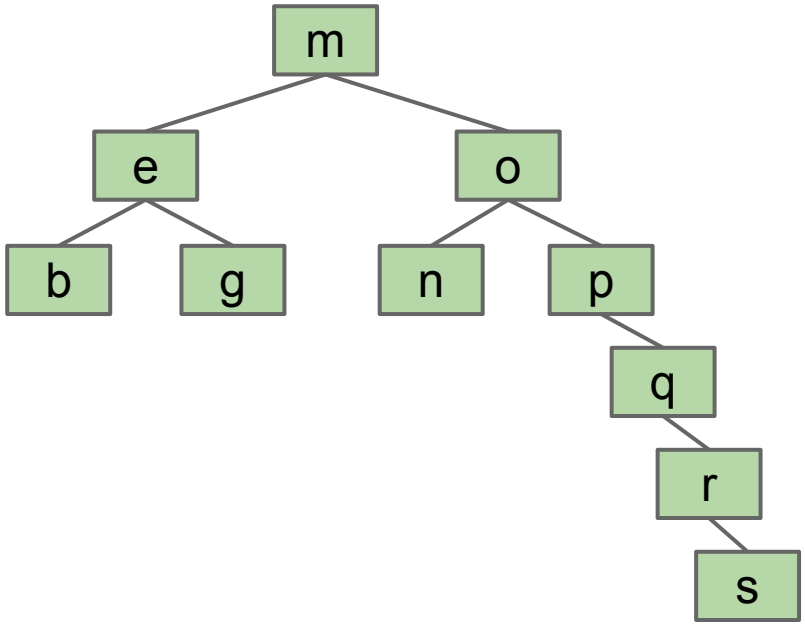
The Trouble with Trees

Last time: BSTs have potential performance issue if they get “spindly” (too tall).

- Worst case: Items inserted in order.
- Fundamental issue: Too lazy about where to store data.

One solution:

- Insert in random order.
  - Results in Θ(log N) height with extremely high probability.
- Why don’t we just do this?



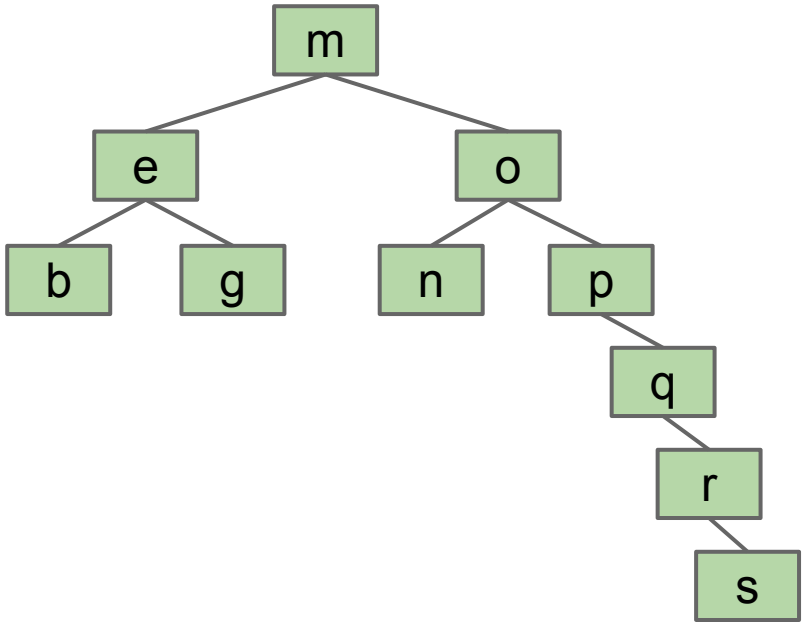
The Trouble with Trees

Last time: BSTs have potential performance issue if they get “spindly” (too tall).

- Worst case: Items inserted in order.
- Fundamental issue: Too lazy about where to store data.

One solution:

- Insert in random order.
  - Results in Θ(log N) height with extremely high probability.
- Why don’t we just do this?
  - Might not have all the data up front.

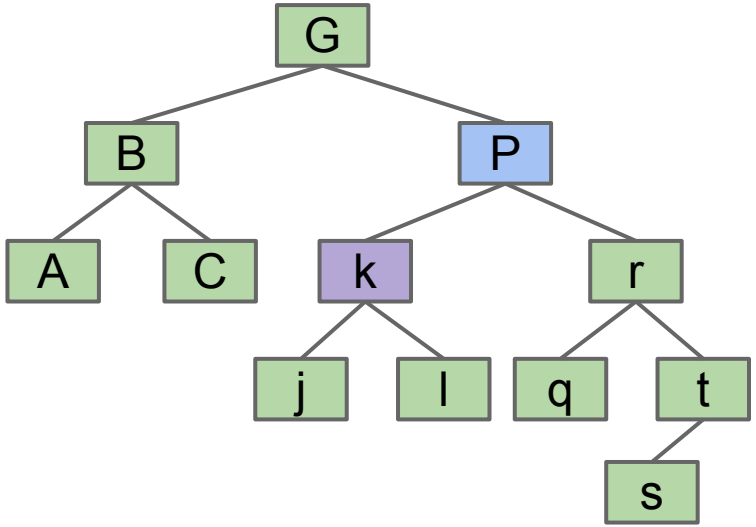


Tree Rotation

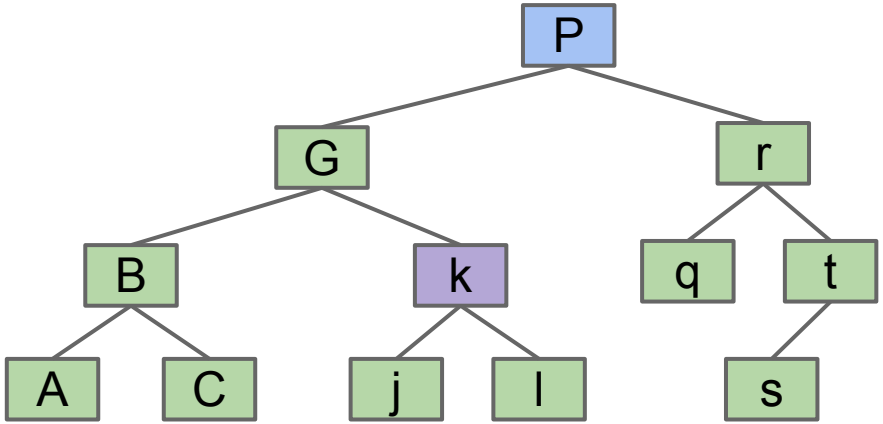
Tree Rotation Example

Suppose we have a search tree as shown below:

- RotateLeft(G): G moves left, **promote its right child in the only natural way**.
  - Promoting P means P is the new parent of G.
- Semantics of the tree are completely unchanged!



To reverse the operation: RotateRight(P)



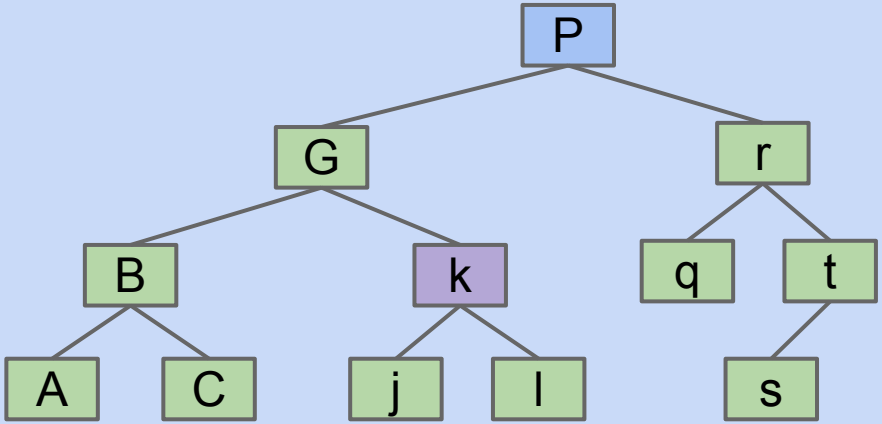
Result of RotateLeft(G):

- Here, rotation reduced height of tree!

Another Tree Rotation Example

Suppose we have a search tree as shown below:

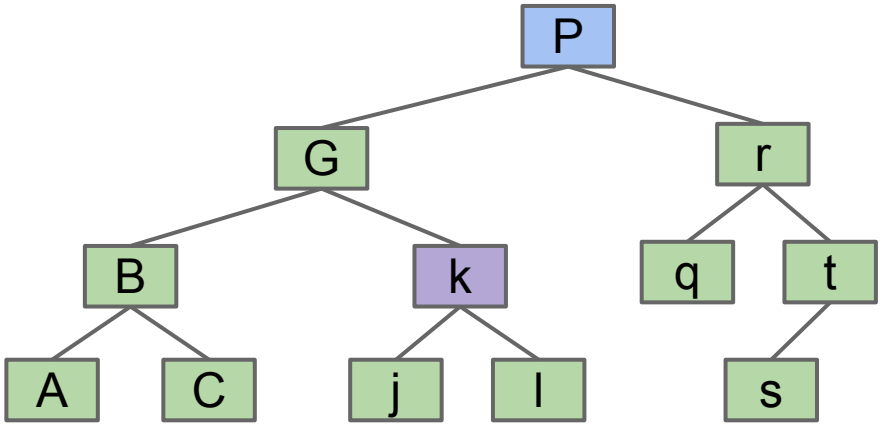
- RotateRight(P): P moves right, **promote its left child in the only natural way**.
- Try to predict the new shape of the tree.

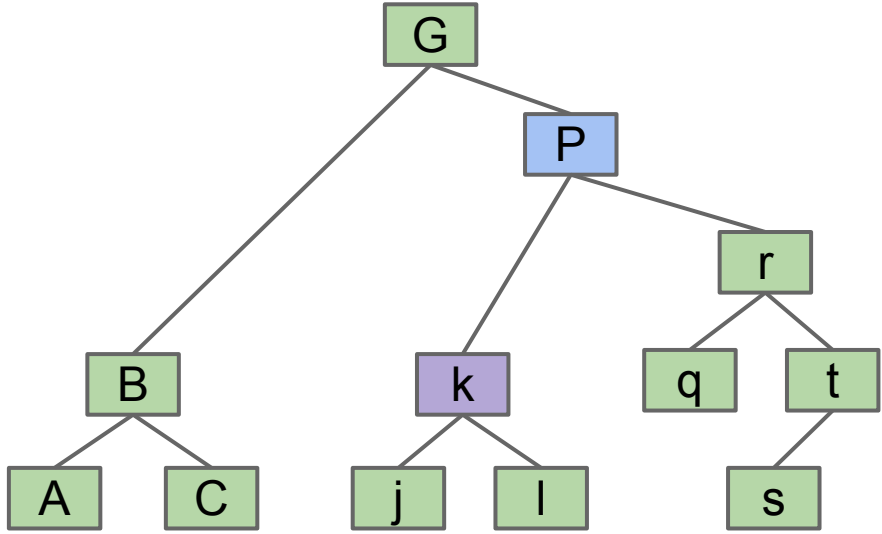


Another Tree Rotation Example

Suppose we have a search tree as shown below:

- RotateRight(P): P moves right, **promote its left child in the only natural way**.
- Try to predict the new shape of the tree.

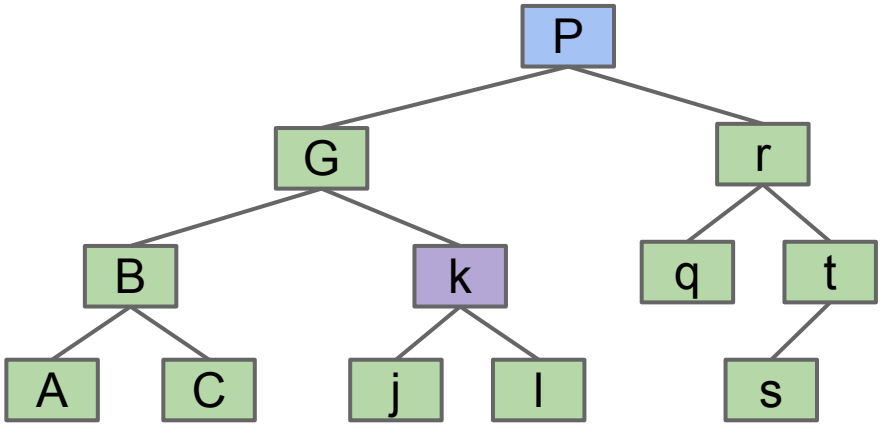


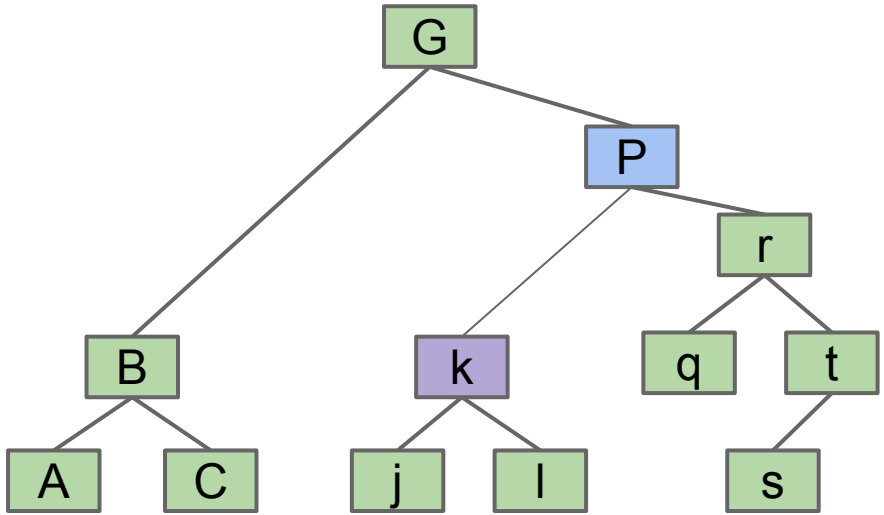


Another Tree Rotation Example

Suppose we have a search tree as shown below:

- RotateRight(P): P moves right, **promote its left child in the only natural way**.
- We see that the operation we did two slides ago has been reversed.





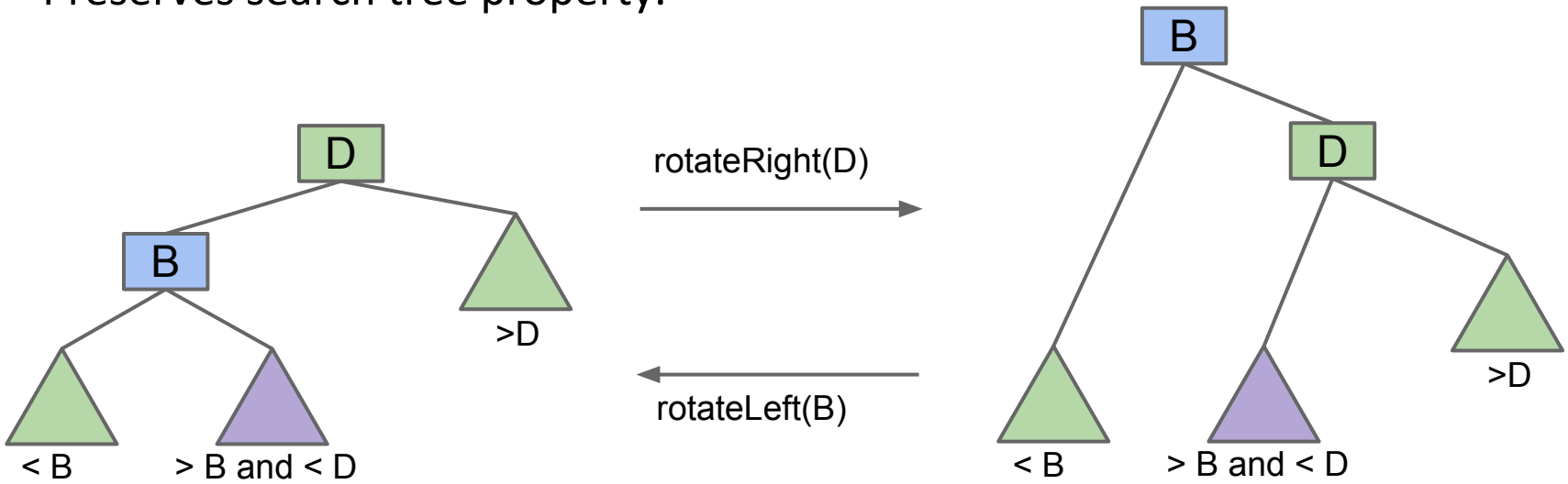
Result of RotateRight(P):

- Here, rotation increased height of tree!

Rotation: An Alternate Approach to Balance

Rotation:

- Can shorten (or lengthen) a tree.
- Preserves search tree property.



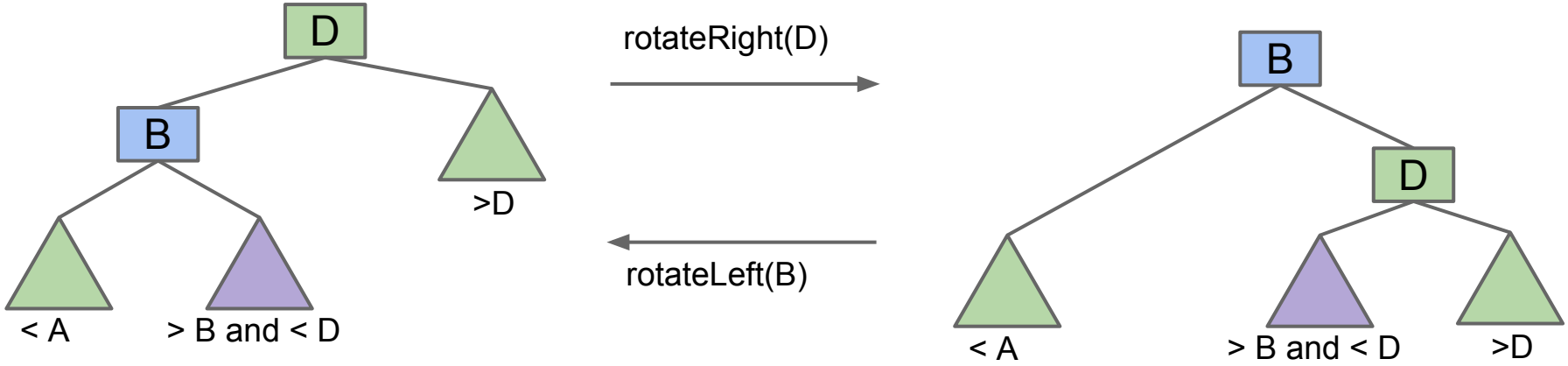
Can use rotation to balance a BST: Demo

- Non-obvious fact: Rotation allows balancing of **any** BST.

Rotation: An Alternate Approach to Balance

Rotation:

- Can shorten (or lengthen) a tree.
- Preserves search tree property.



One way to *achieve* balance:

- Rotate after each insertion and deletion to *maintain* balance.
- … the mystery is to know which rotations. We'll come back to this.

B-trees / 2-3 trees / 2-3-4 trees

Search Trees

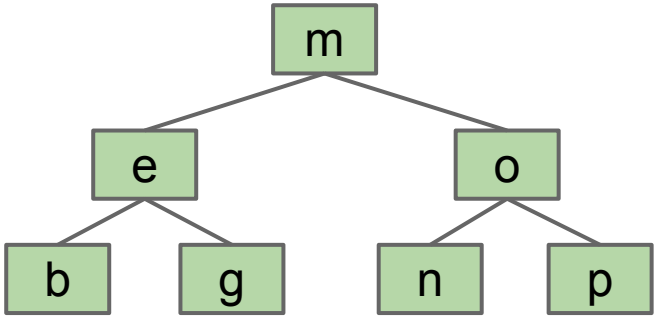
There are many types of search trees:

- **Binary search trees**: Require rotations to maintain balance. There are many strategies for rotation (AVL, weight-balancing, **red-black**).
- Treaps.
- Splay trees.
- **2-3 / 2-3-4 trees / B-trees**: No rotations required.
  - We'll develop this idea together in the following slides.

Intriguingly, a deep understanding of 2-3 trees also provides a rotation strategy for BSTs called "red-black".

A Weirder Solution

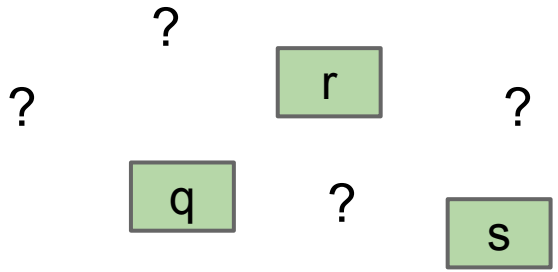
The problem is adding new leaves. Could balance with rotation but it's not obvious which rotations to perform.



Weirder solution: Never add new leaves.

- Tree can never get imbalanced.

Q: What do we do with incoming keys?

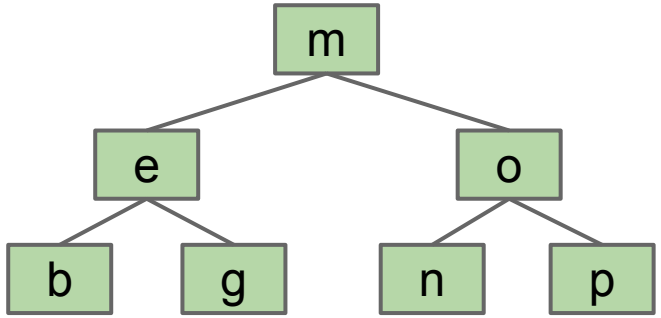


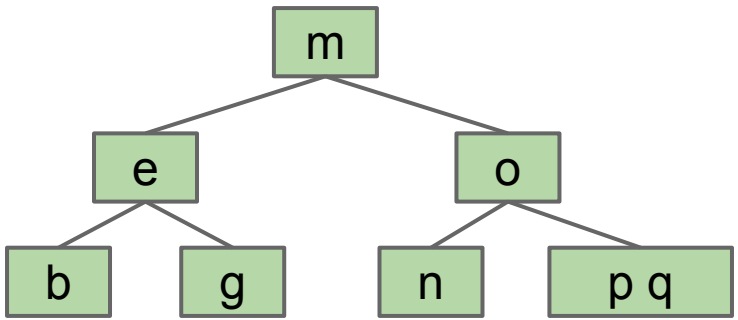
A Weirder Solution

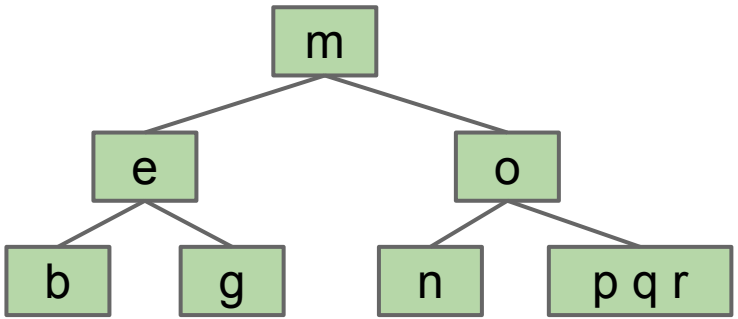
The problem is adding new leaves. Could balance with rotation but it's not obvious which rotations to perform.

Weirder solution: Overstuff the leaf nodes.

- "Overstuffed tree" always has balanced height, because leaf depths never change.
  - Height is just max(depth).



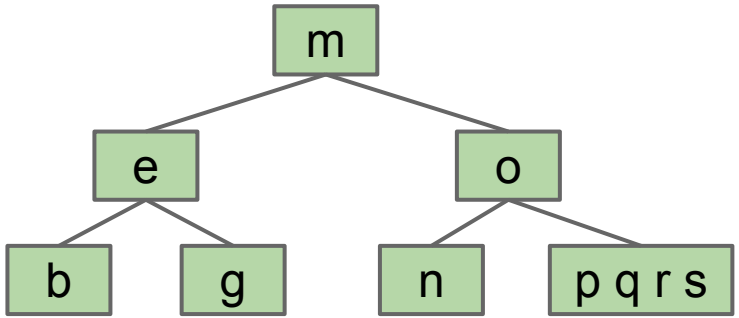




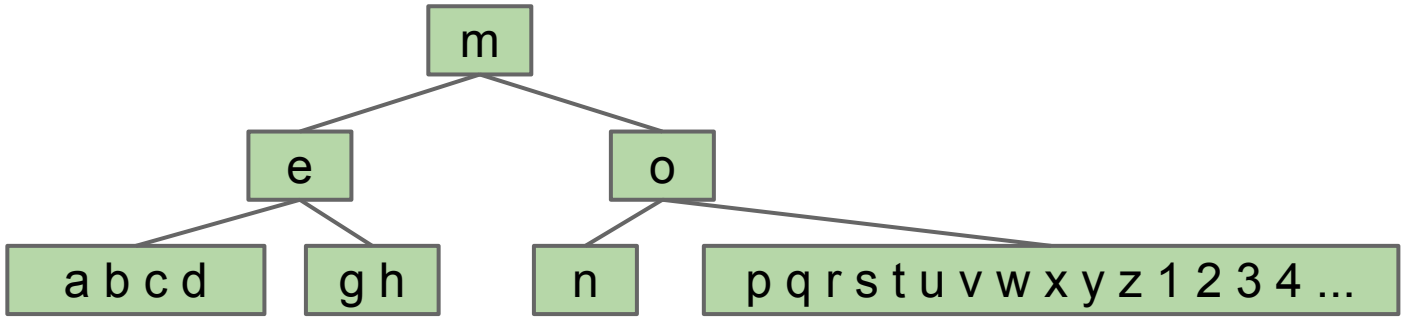
A Weirder Solution: The “Overstuffed Tree”

This is a logically consistent but very weird data structure.

- contains(r):
  - Search for the appropriate leaf.
  - Search the leaf for ‘r’



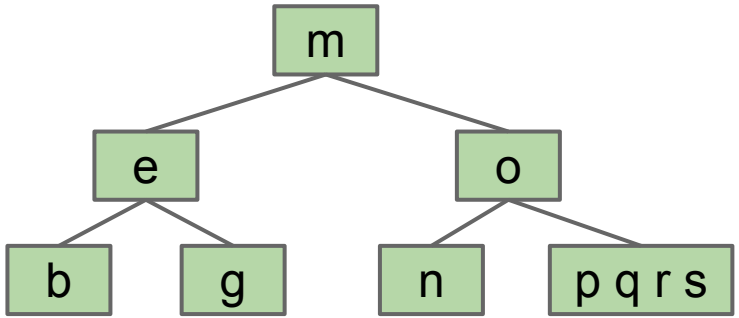
Q: What is the problem with this idea?



Revising Our Overstuffed Tree Approach

New problem:

- Leaf nodes can get too juicy.



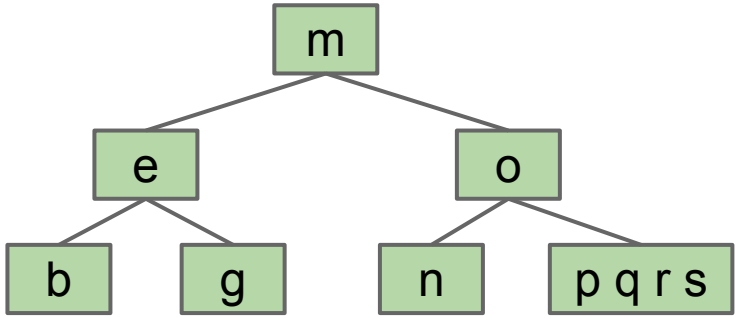
Q: Solution?

- Goal: Still want no new leaf nodes.

Revising Our Overstuffed Tree Approach

New problem:

- Leaf nodes can get too juicy.

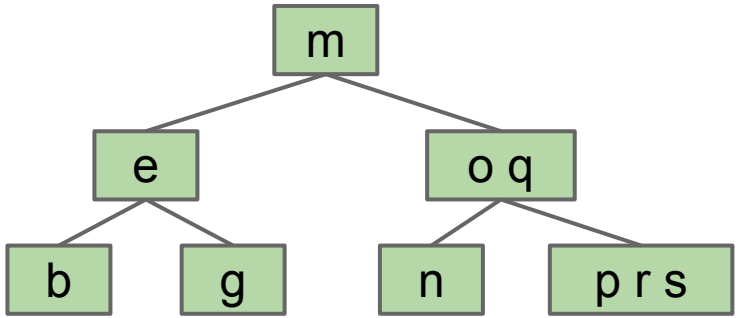


Solution?

- Set a cap on the number of items, say 3.
- If more than cap, give an item to parent.
  - Which one? Let's say the left-middle.

Q: What's the problem now?

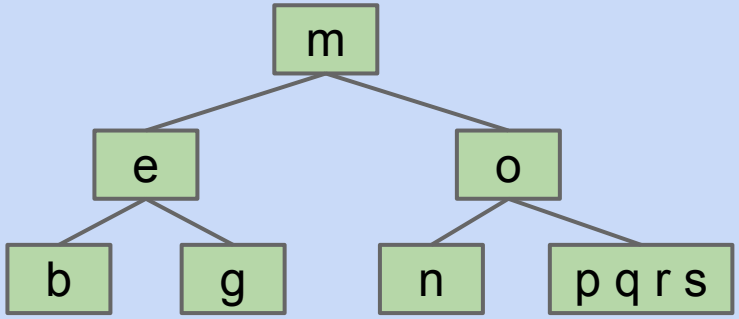
- Items between o and q have no home.



Revising Our Overstuffed Tree Approach

New problem:

- Leaf nodes can get too juicy.

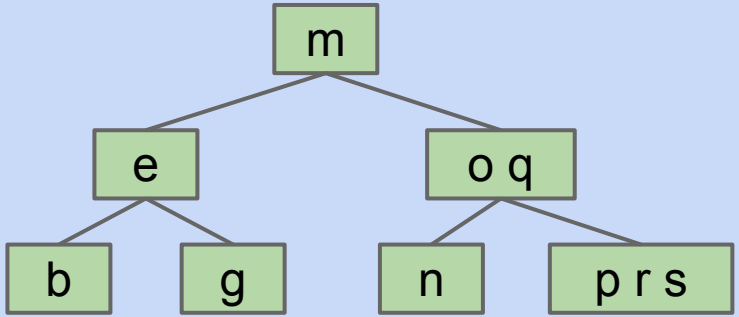


Solution?

- Set a cap on the number of items, say 3.
- If more than cap, give an item to parent.
  - Which one? Let's say the left-middle.

Challenge for you:

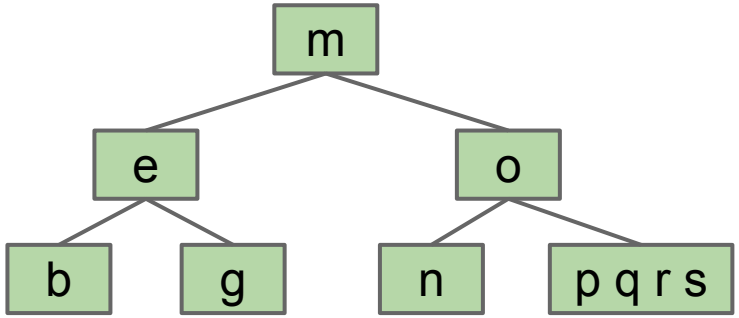
- How can we tweak this idea to make it work?



Revising Our Overstuffed Tree Approach [your answer]

New problem:

- Leaf nodes can get too juicy.

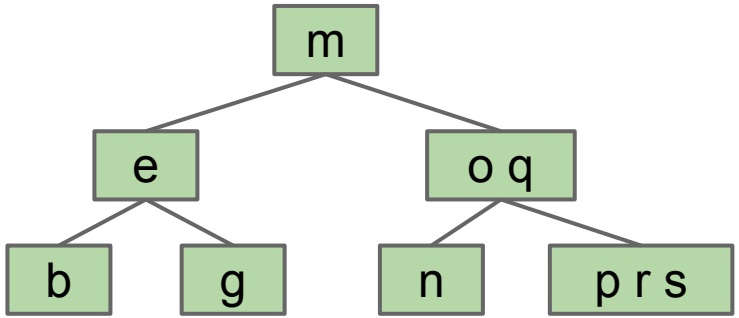


Solution?

- Set a cap on the number of items, say 3.
- If more than cap, give an item to parent.
  - Which one? Let's say the left-middle.

Challenge for you:

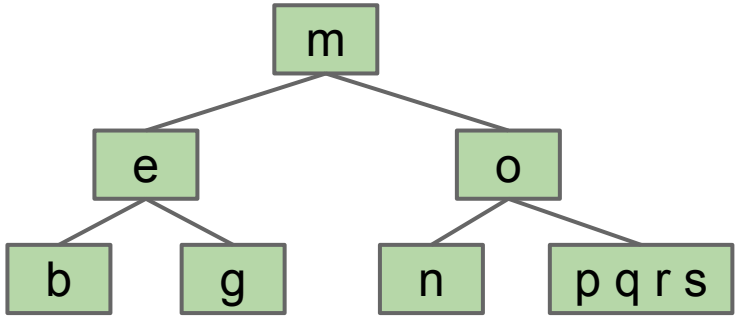
- How can we tweak this idea to make it work?



Revising Our Overstuffed Tree Approach: Node Splitting

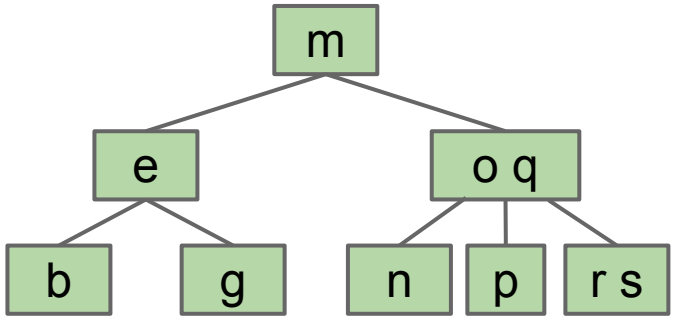
New problem:

- Leaf nodes can get too juicy.



Solution?

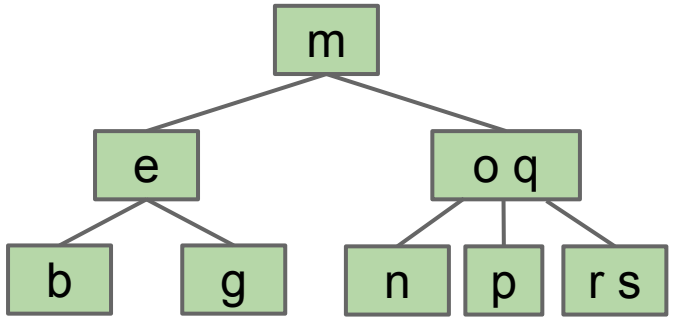
- Set a cap on the number of items, say 3.
- If more than cap, give an item to parent.
  - Pulling item out of juicy node splits it into left and right.
  - Parent node now has three children!



Revising Our Overstuffed Tree Approach: Node Splitting

This is a logically consistent and not so weird data structure.

- contains(r):
  - r > m, so go right
  - r > o, so compare vs. q
  - r > q, so go right



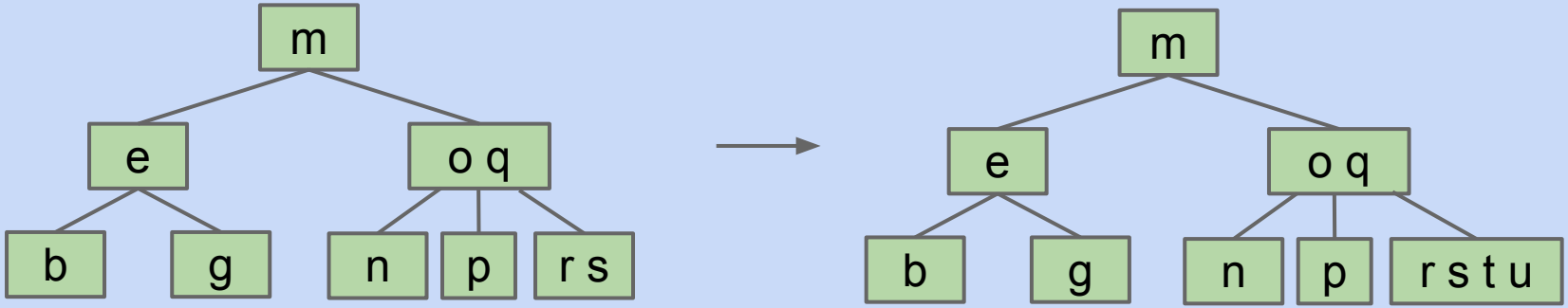
Examining a node might cost us Z compares, but that's OK since Z is capped.
(where Z is the # of items in a node)

What if a non-leaf node gets overstuffed? Can we split that?

- Sure, we'll do this in a few slides, but first...

Insertion Understanding Check

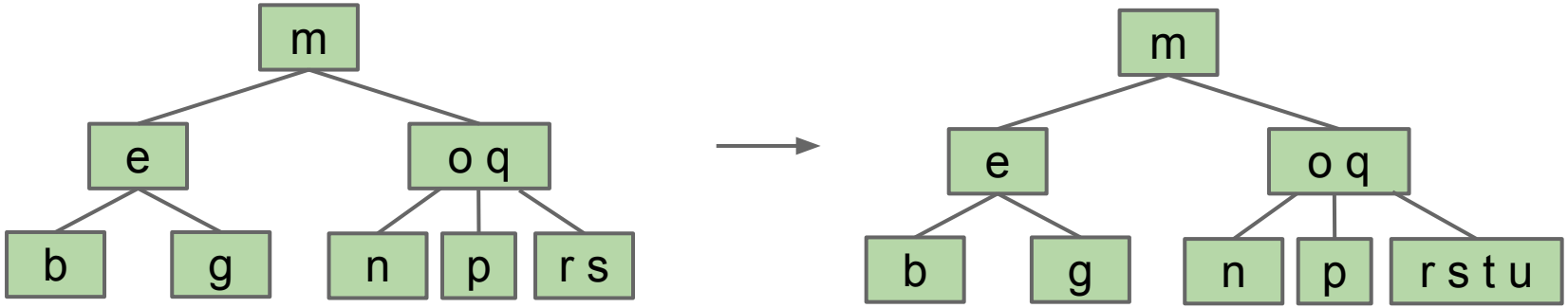
- Suppose we insert t, u:



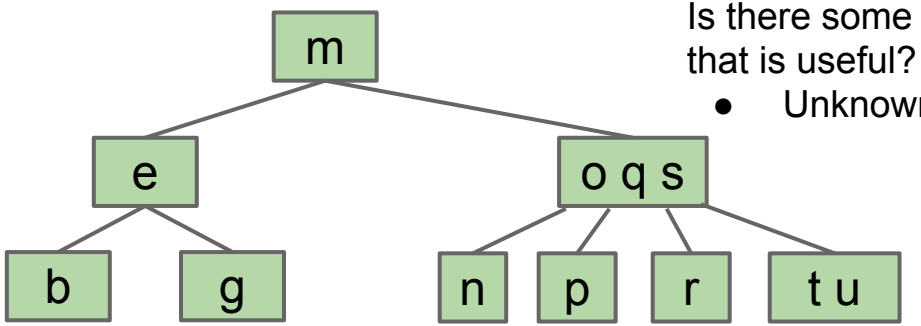
- Q: If our cap is at most 3 items per node, draw post-split tree:

Insertion Understanding Check

- Suppose we insert t, u:



- Q: If our cap is at most 3 items per node, draw post-split tree:

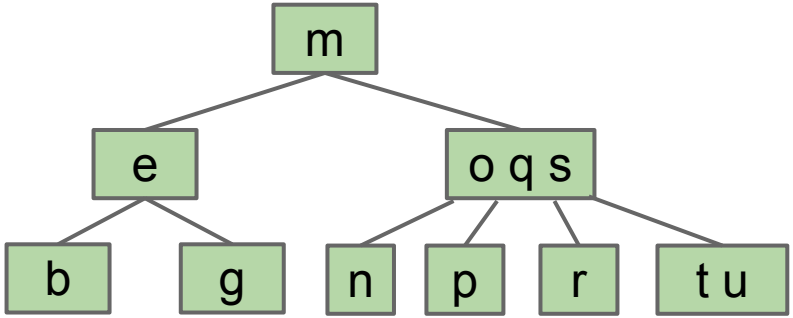


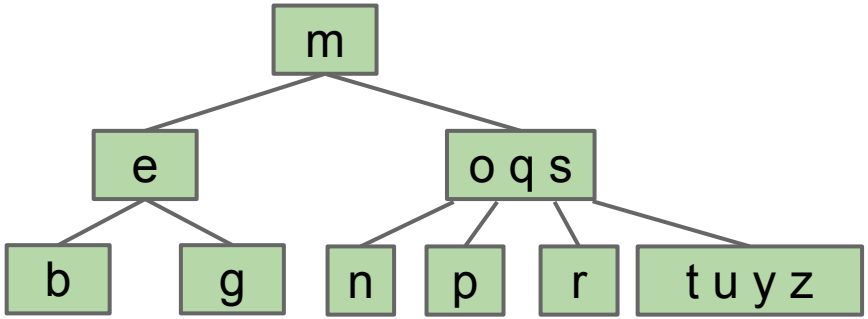
Is there some send up / send down variant that is useful?

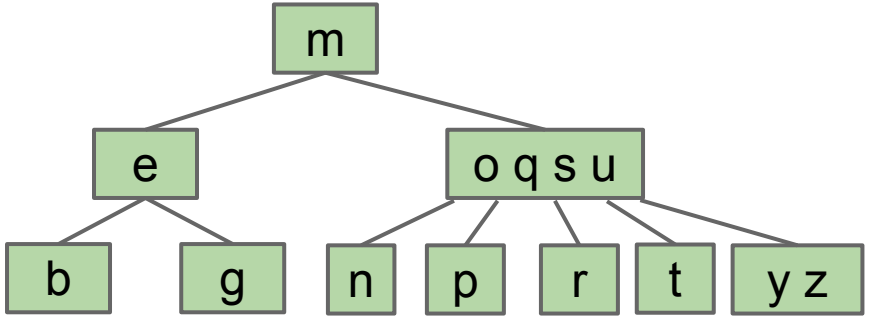
- Unknown, but more complicated.

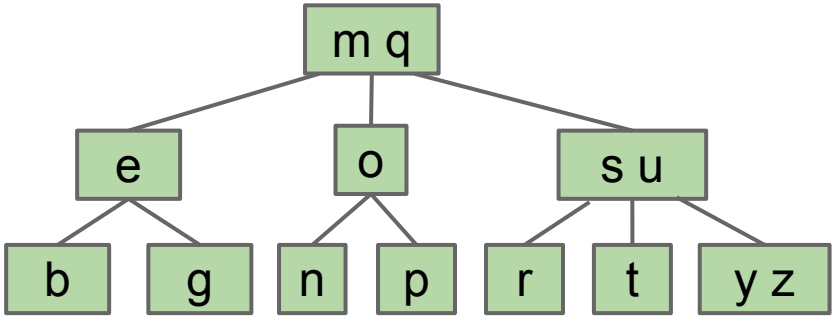
Insertion: Chain Reaction

- Suppose we insert y, z:

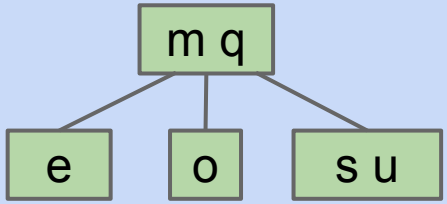


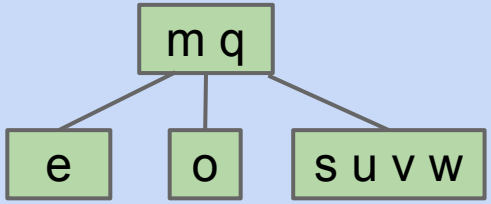


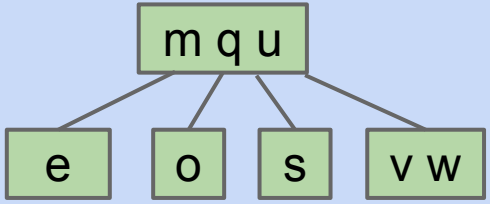


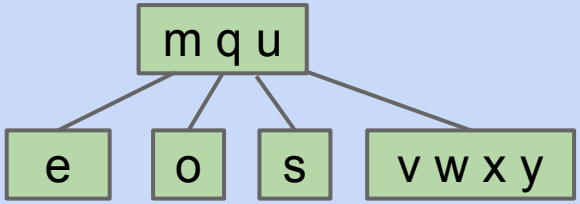


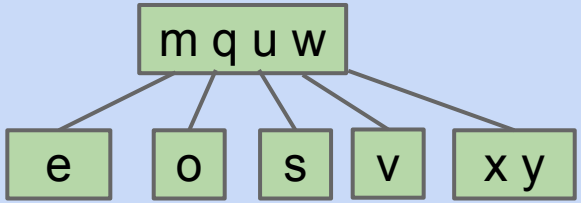
What Happens If The Root Is Maxed Out on Juice?





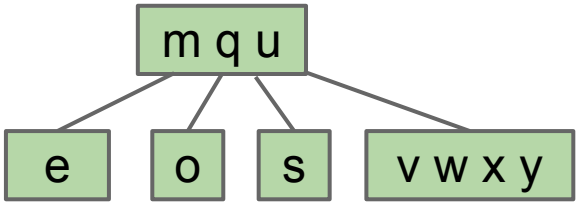


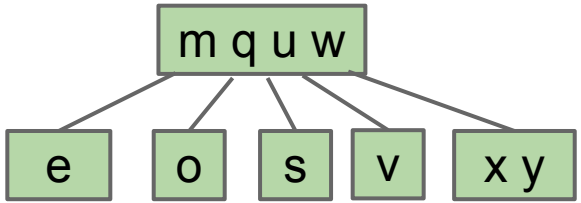




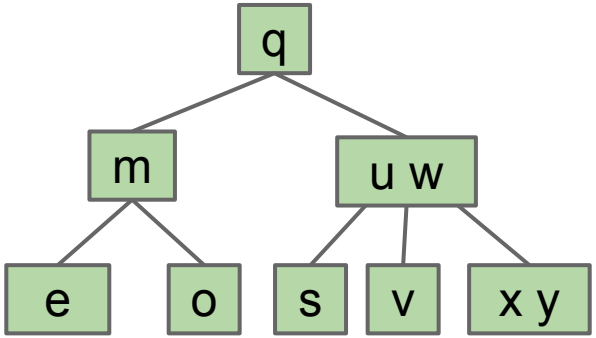
Challenge: Draw the tree after the root is split.

What Happens If The Root Is Maxed Out on Juice?





Challenge: Draw the tree after the root is split.



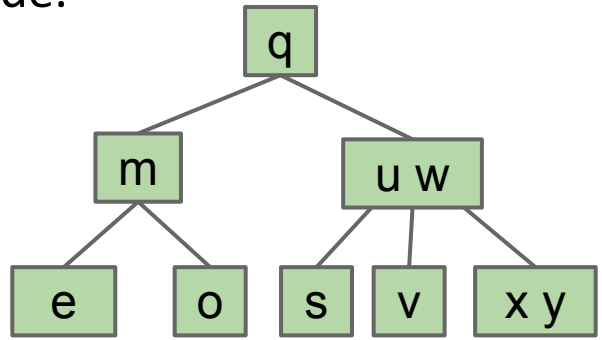
Perfect Balance

Observation: Splitting-trees have perfect balance.

- If we split the root, every node gets pushed down by exactly one level.
- If we split a leaf node or internal node, the height doesn't change.

Bottom line: All operations have guaranteed Θ(log N) time.

- More details on next slide.



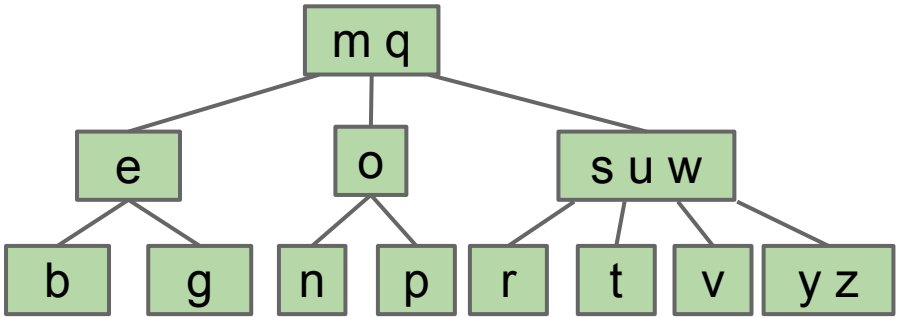
Perfect Balance and Logarithmic Height

M: Max number of children (one more than the item cap).

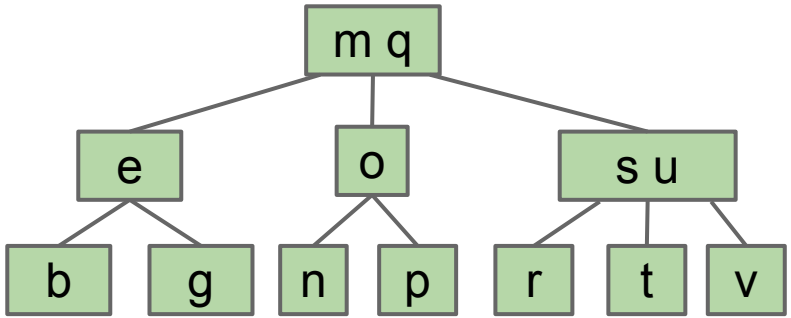
Height: Between $\log_M(N)$ and $\log_2(N)$

Max number of splitting operations per insert: ~H

Time per insert/contains: Θ(H), or equivalently Θ(log N)



For M = 4:
Max 3 items per node.
Max 4 non-null children per node.



For M = 3:
Max 2 items per node.
Max 3 non-null children per node.

The Real Name for Splitting Trees is “B Trees”

Splitting tree is a better name, but I didn’t invent them, so we’re stuck with their real name: B-trees.

- A B-tree of order M=4 (like we used today) is also called a 2-3-4 tree or a 2-4 tree.
  - The name refers to the number of children that a node can have, e.g. a 2-3-4 tree node may have 2, 3, or 4 children.
- A B-tree of order M=3 (like in the textbook) is also called a 2-3 tree.

The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

- Douglas Corner (The Ubiquitous B-Tree)

A note on Terminology

B-Trees are most popular in two specific contexts:

- Small M (M=3 or M=4):
  - Used as a conceptually simple balanced search tree (as today).
- M is very large (say thousands).
  - Used in practice for databases and filesystems (i.e. systems with very large records).
    - See CS186.

Note: Since the small M tree isn't practically useful (as we'll see in a few slides), people usually only use the name B-tree to refer to the very large M case.

Red-Black Trees

The Bad News

2-3 trees (and 2-3-4 trees) are a real pain to implement, and suffer from performance problems. Issues include:

- Maintaining different node types.
- Interconversion of nodes between 2-nodes and 3-nodes.
- Walking up the tree to split nodes.

```
public void put(Key key, Value val) {
   Node x = root;
   while (x.getTheCorrectChild(key) != null) {
      x = x.getTheCorrectChildKey();
      if (x.is4Node()) x.split();
   }
   if      (x.is2Node()) x.make3Node(key, val);
   else if (x.is3Node()) x.make4Node(key, val);
}
```

“Beautiful algorithms are, unfortunately, not always the most useful” - Knuth

Search Trees

There are many types of search trees:

- **Binary search trees**: Require rotations to maintain balance. There are many strategies for rotation. Coming up with a strategy is hard.
- **2-3 trees**: No rotations required.

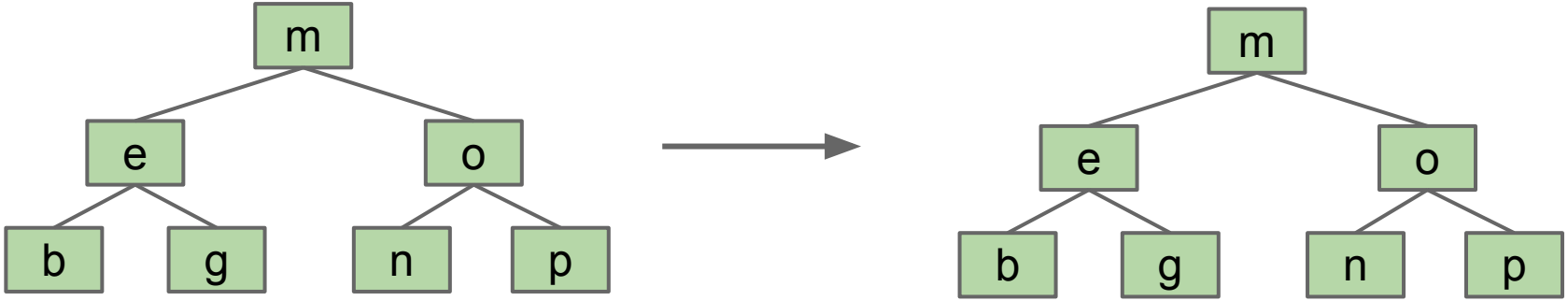
Clever (and strange idea): Build a BST that is isometric (structurally identical) to a 2-3 tree.

- Use rotations to ensure the isometry.
- Since 2-3 trees are balanced, rotations on BST will ensure balance.

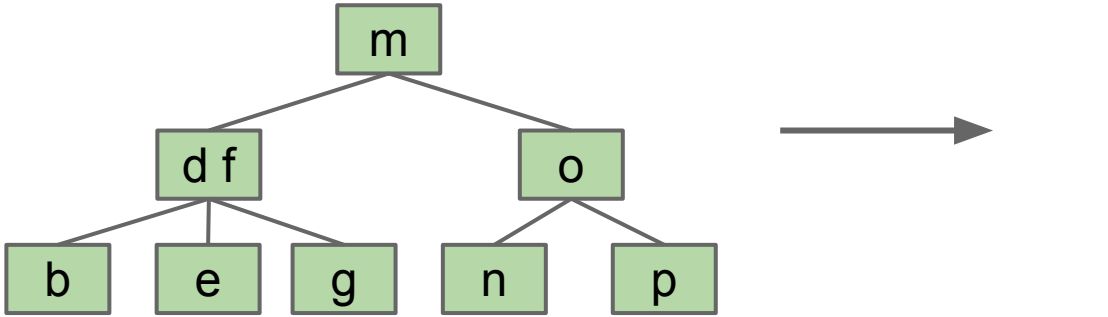
Representing a 2-3 Tree as a BST

A 2-3 tree with only 2 nodes is trivial.

- BST is exactly the same!



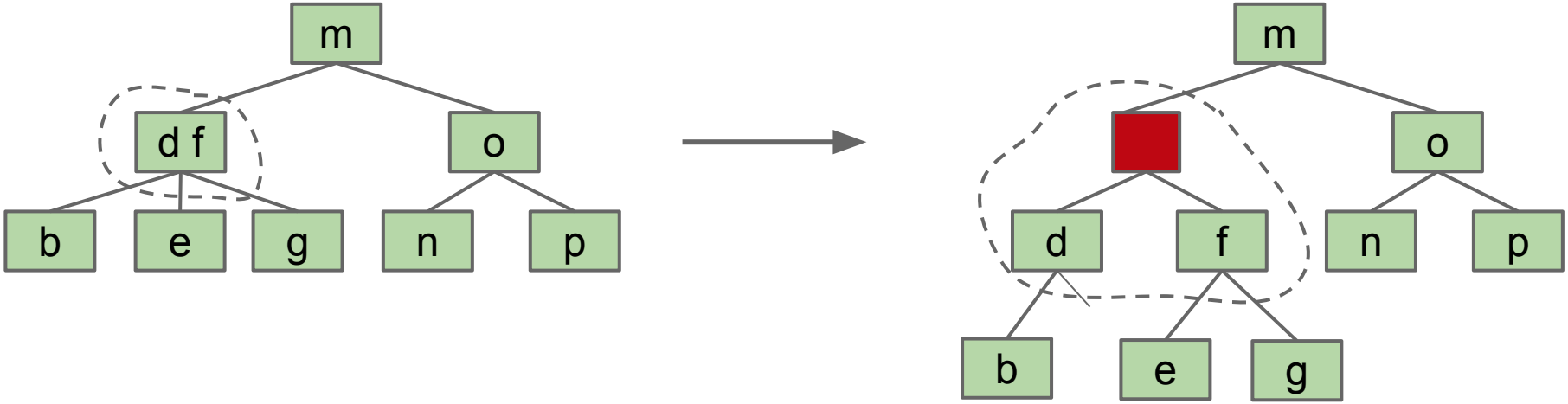
What do we do about 3 nodes?



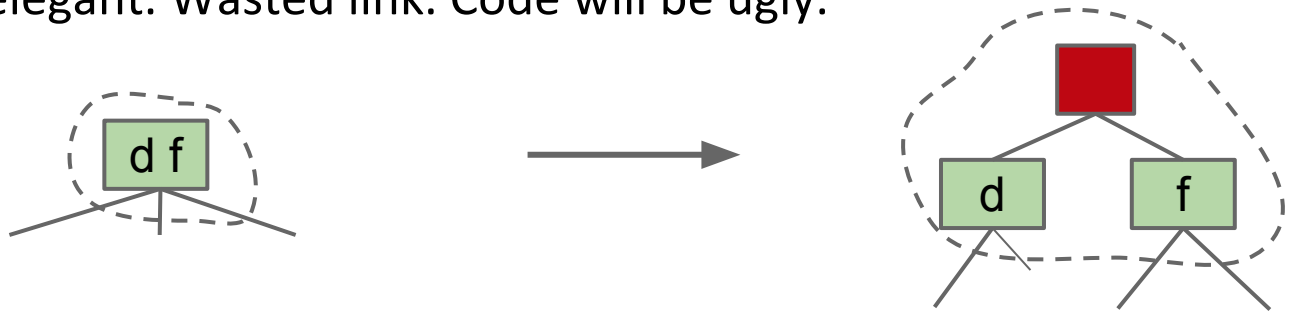
????

Representing a 2-3 Tree as a BST: Dealing with 3-Nodes

Possibility 1: Create dummy “glue” nodes.

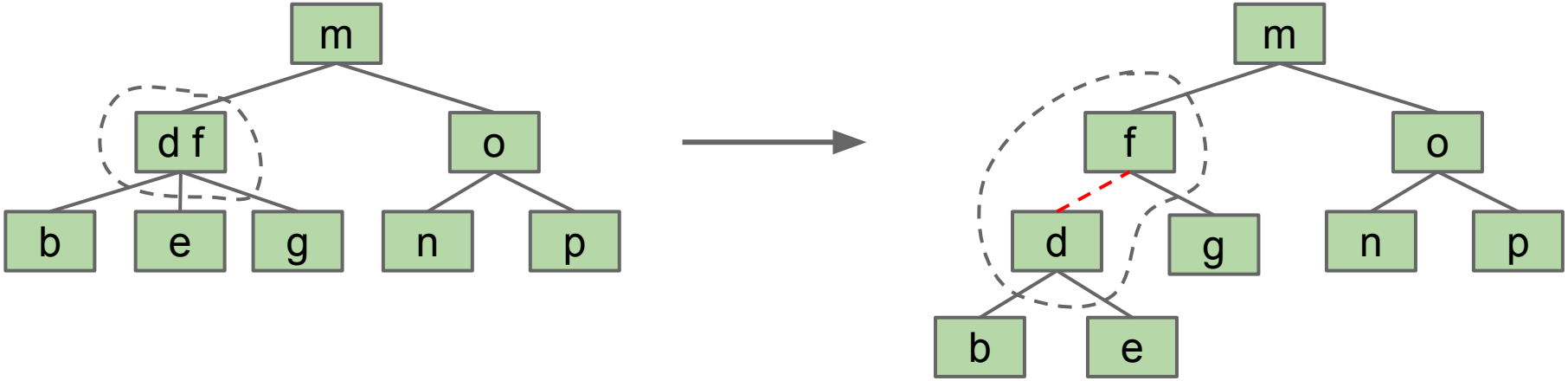


Result is inelegant. Wasted link. Code will be ugly.

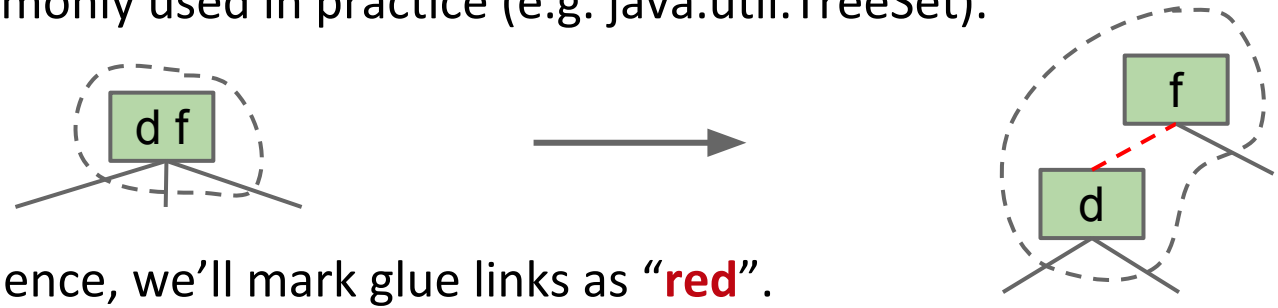


Representing a 2-3 Tree as a BST: Dealing with 3-Nodes

Possibility 2: Create "glue" links with the smaller item **off to the left**.



Idea is commonly used in practice (e.g. java.util.TreeSet).

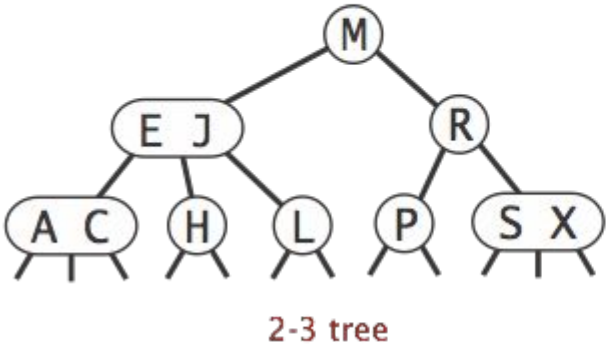


For convenience, we'll mark glue links as "**red**".

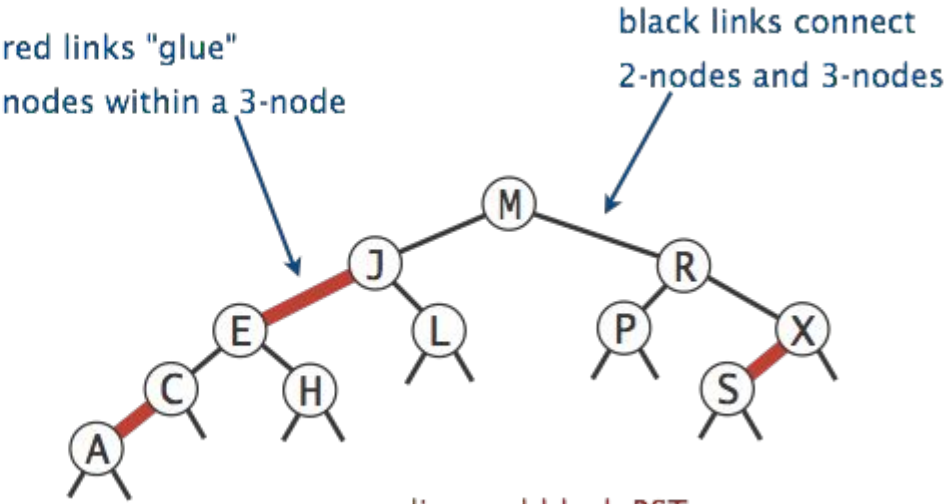
Left-Leaning Red Black Binary Search Tree (LLRB)

Any BST that maintains an isometry with a 2-3 tree has these properties:

- No node has two red links (otherwise it'd be like a 4 node).
- Every path from root to a leaf has same number of black links.
- Red links lean left (**idiosyncratic to our textbook**), see [Red-Black Tree].
- Also called a "left leaning red black binary search tree (LLRB)".



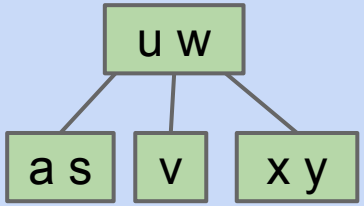
2-3 tree



corresponding red-black BST

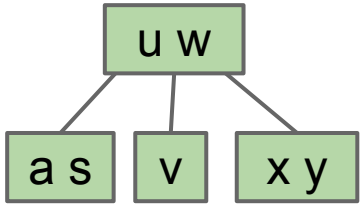
Left-Leaning Red Black Binary Search Tree (LLRB)

Draw the LLRB corresponding to the 2-3 tree shown below.



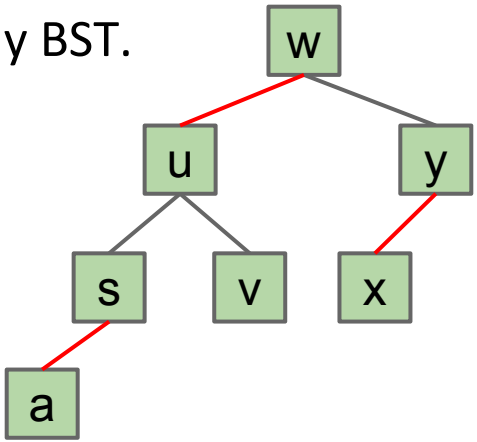
Left-Leaning Red Black Tree (LLRB)

Draw the LLRB corresponding to the 2-3 tree shown below.



Searching an LLRB tree for a key is easy.

- Treat it exactly like any BST.



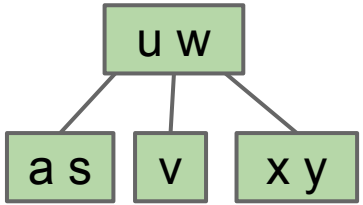
Left-Leaning Red Black Tree (LLRB)

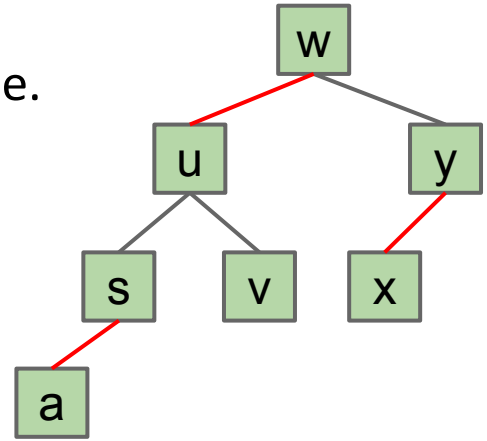
Facts of interest:

- For any 2-3 tree (which is balanced), there exists a corresponding red-black tree that has depth no more than 2 times the depth of the 2-3 tree.

With a HUGE dose of cleverness can use rotations to maintain isometry after each insertion and deletion.

- Rather advanced topic.
- You don't need to memorize the rules that follow this slide.





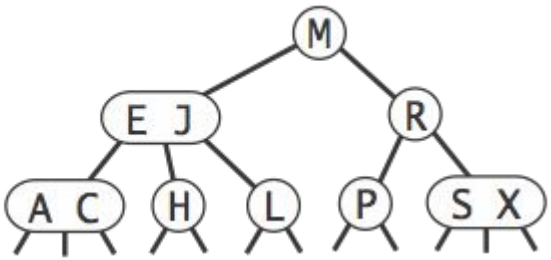
Maintaining Isometry Through Rotations (Optional)

The Isometry

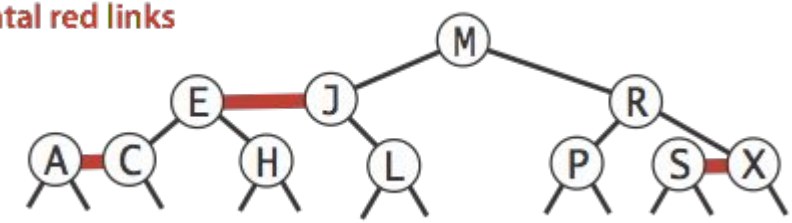
There exists an isometry between:

- 2-3 Tree
- LLRB

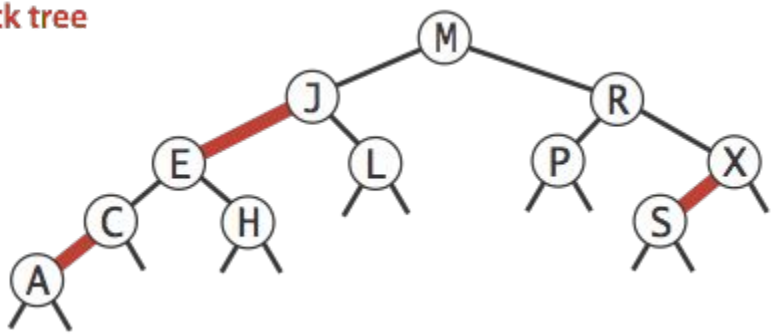
2-3 tree



horizontal red links



red–black tree

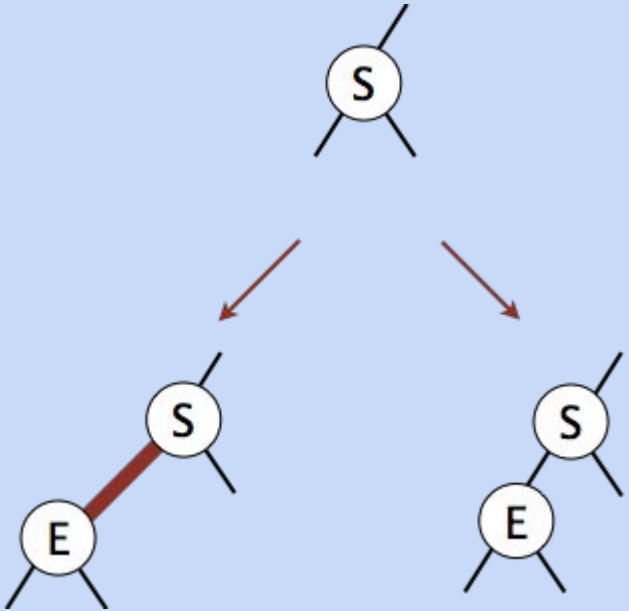


Implementation of an LLRB is based on maintaining this isometry.

- When performing LLRB operations, pretend like you're a 2-3 tree.
- Preservation of isometry will involve tree rotations.

Isometry Maintenance

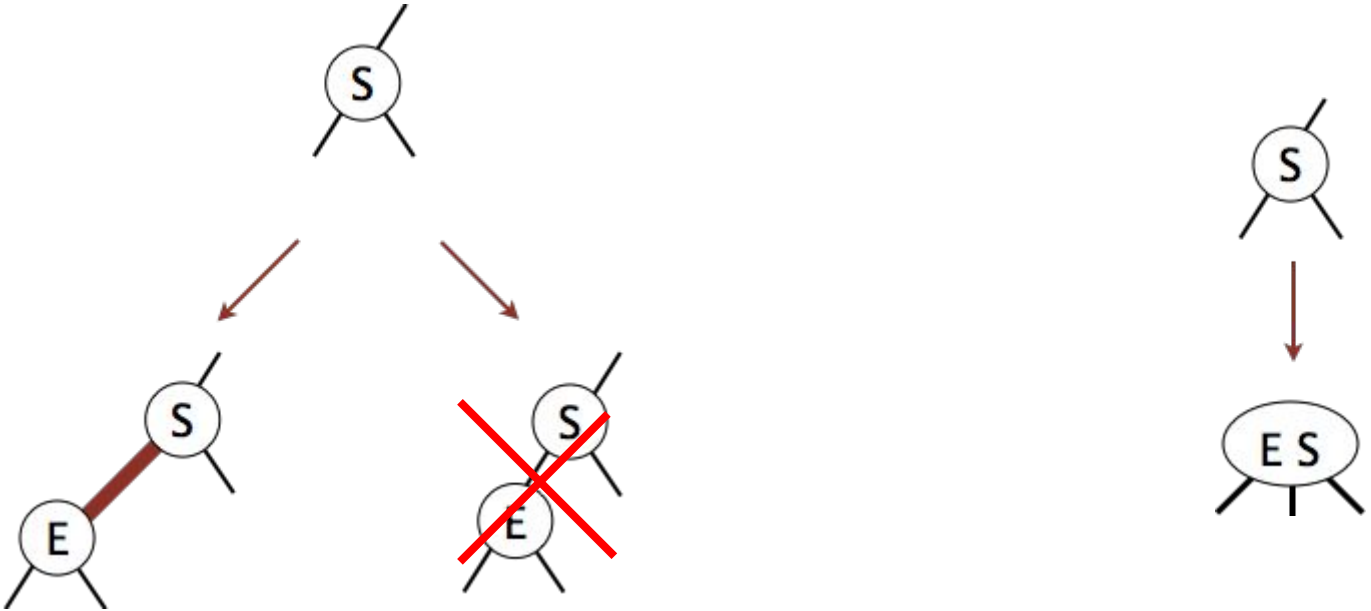
Should we use a red or black link when inserting?



Isometry Maintenance

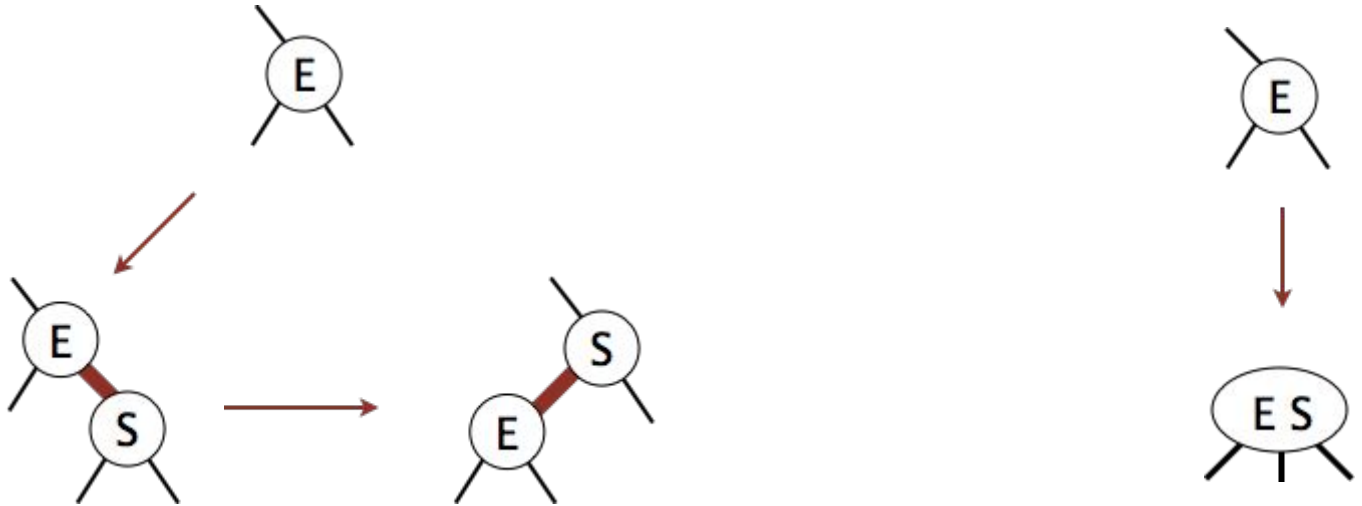
Should we use a red or black link when inserting?

- Use red! In 2-3 trees we ALWAYS start by increasing node size.



Isometry Maintenance (Non-Trivial Case 1: Right-Insert)

Suppose we have leaf E, and insert S with a red link. What is the problem below?

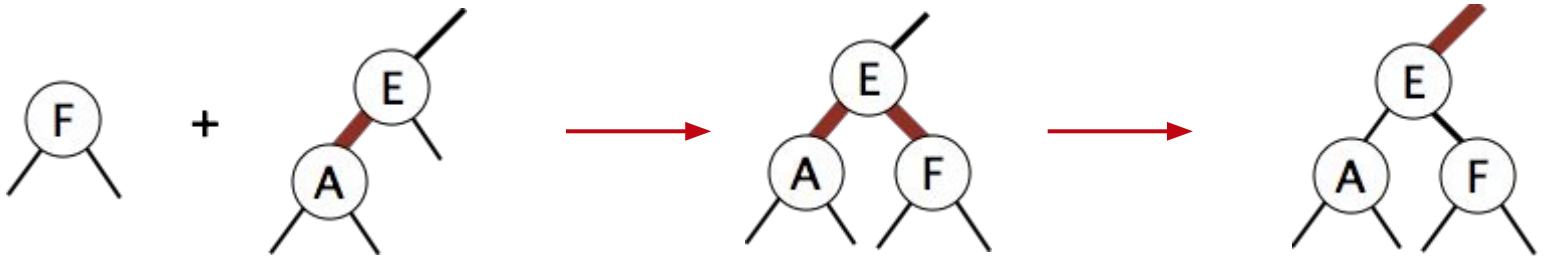


How do we fix this?

- Swap roles of S and E, i.e. rotateLeft(E).

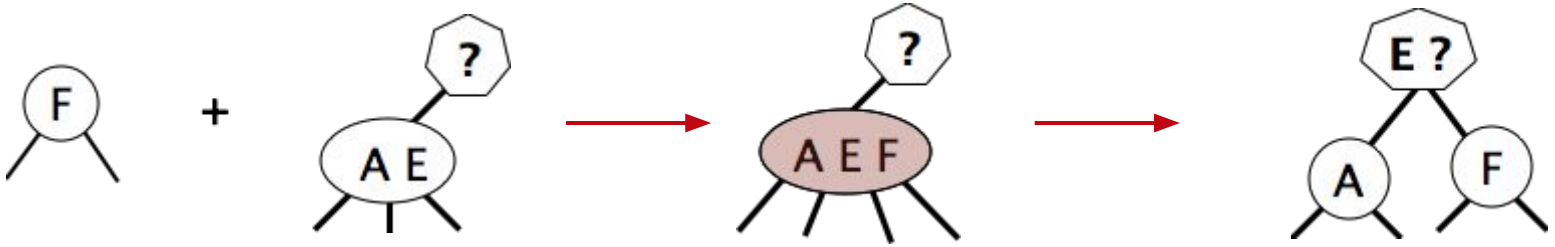
Isometry Maintenance (Non-Trivial Case 2: Two Red Children)

Suppose we add F to an LLRB tree containing E and A as shown below:



Flip Colors.

What do we do? WW23TD?

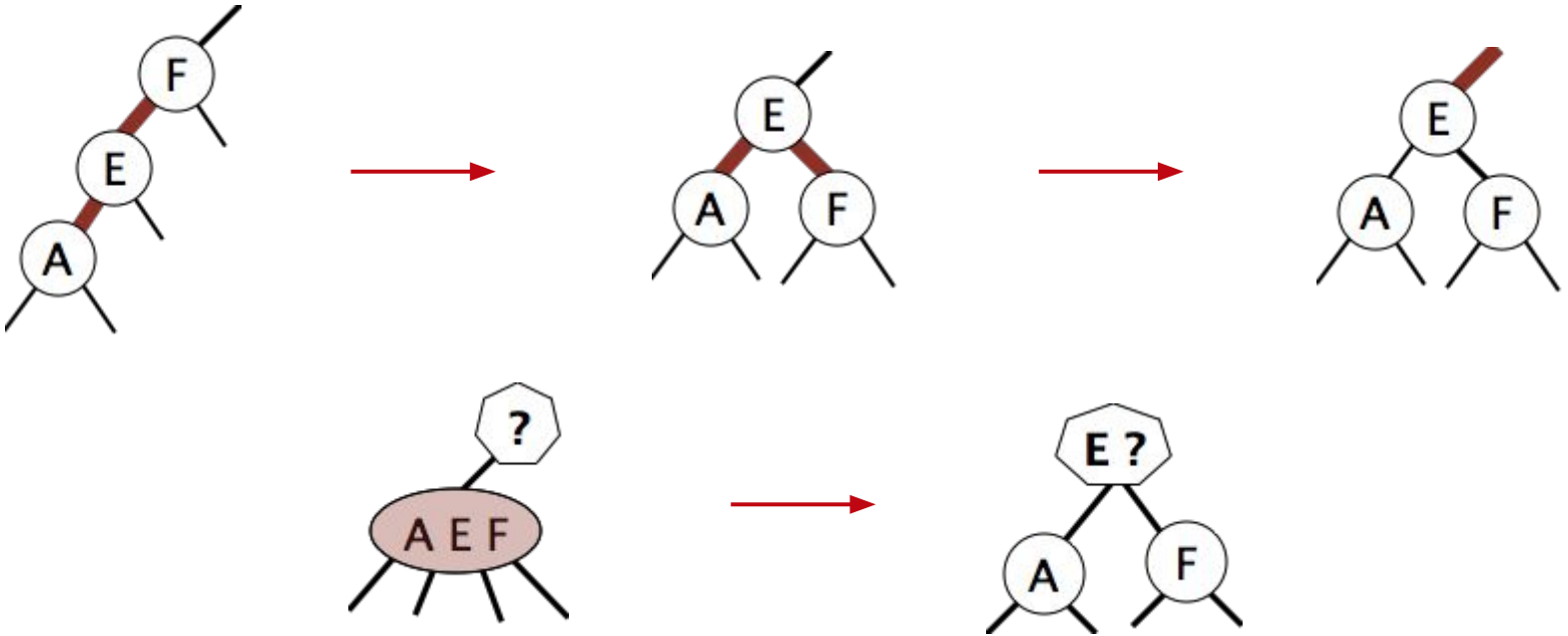


Split.

Isometry Maintenance (Non-Trivial Case 3: Two Reds-in-a-Row)

Suppose we add A to E-F. To resolve:

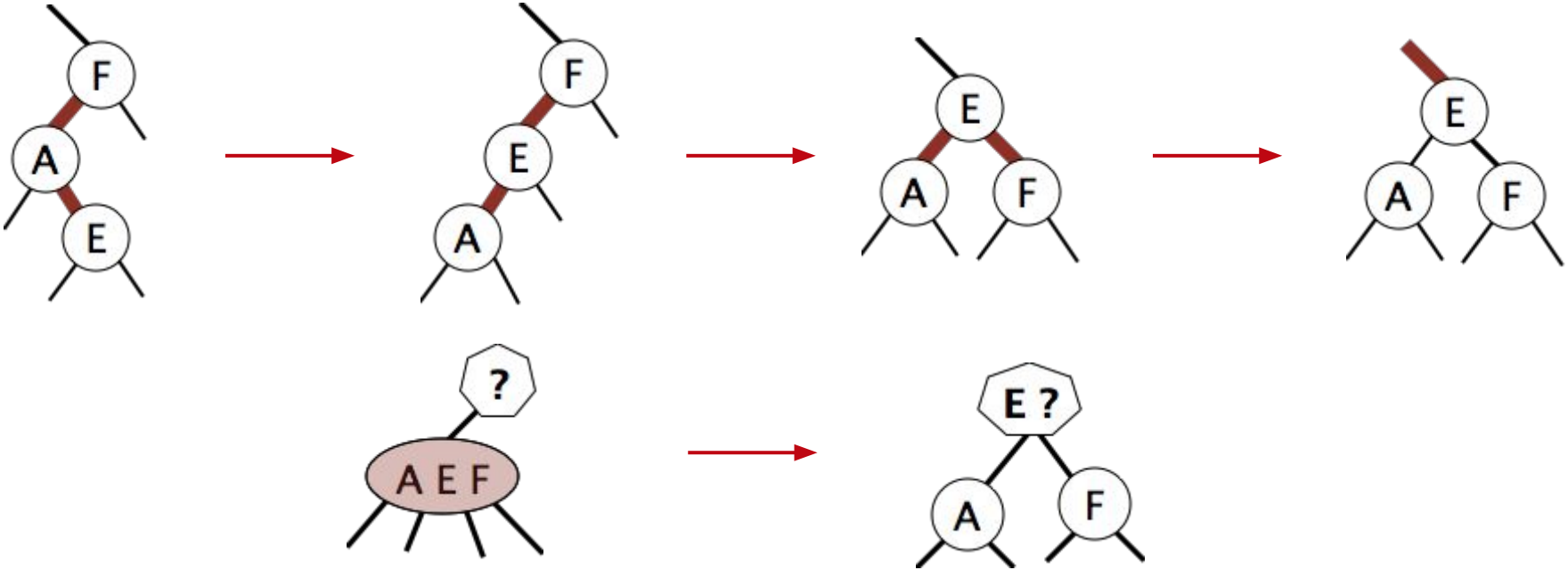
- rotateRight(F), putting us back in the two-red-children case.
- Then color flip.



Isometry Maintenance (Case 1 & 3: Left-Red-Right-Red)

Suppose we add E to F-A. To resolve:

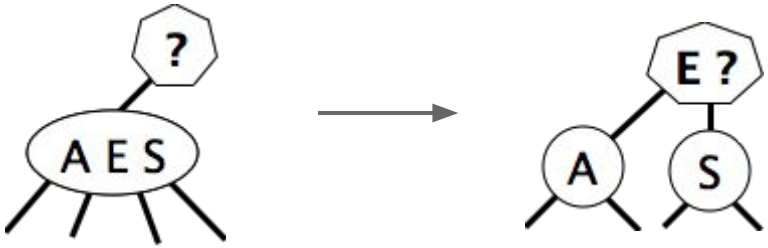
- rotateLeft(A), putting us back in the two-reds-in-a-row
- rotateRight(F), putting us in the two-red children case. Then flip.



Preserving the Isometry After Addition Operations

Violations for 2-3 trees:

- Existence of 4-nodes (bad!)

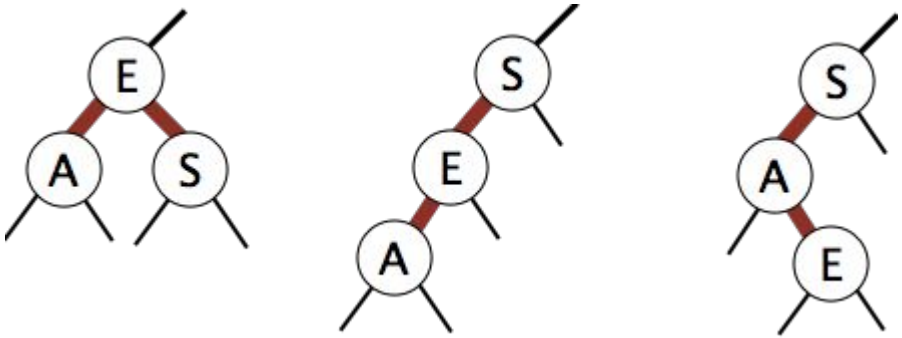


Operations for Fixing 2-3 Tree Violations:

- Splitting a 4-node.

Violations for LLRBs:

- Two red children.
- Two consecutive red links.
- Right red child.

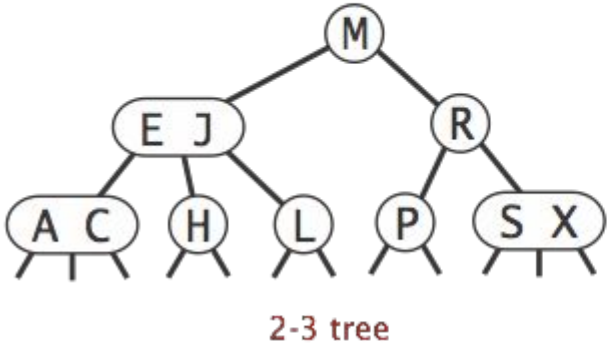


Operations for Fixing LLRB Tree Violations: Tree rotations and Color Flips!

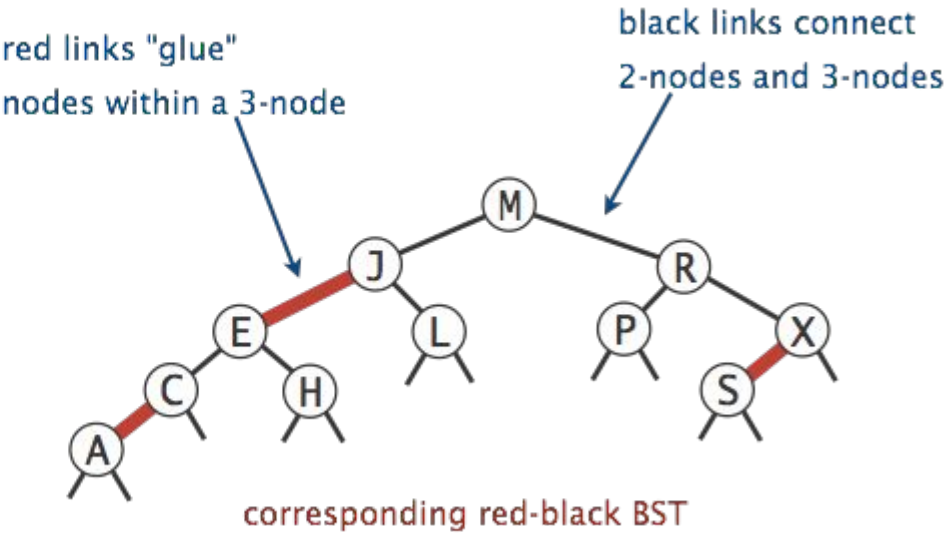
LLRB Performance

In an LLRB, **every path from root to a null has same number of black links.**

Since 2-3 trees are perfectly balanced, LLRBs are **perfectly black balanced**.
Guaranteed logarithmic performance for insert (see textbook for delete).



2-3 tree



corresponding red-black BST

Summary

2-3 and 2-3-4 Trees have perfect balance.

- Height is guaranteed logarithmic.
- After insert/delete, at most 1 split operation per level of the tree.
  - Since height is logarithmic, we have O(log N) splits.
  - insert/delete are therefore O(log N).
- Hard to implement.

LLRBs mimic 2-3 tree behavior using color flipping (and tree rotation).

- Height is guaranteed logarithmic.
- After insert/delete, at most 1 color flip or rotation per level of the tree.
  - Since height is logarithmic, we have O(log N) flips/rotations.
  - insert/delete operations are therefore O(log N).
- Easier to implement, constant factor faster than 2-3 or 2-3-4 tree.

Citations

Bee-tree from https://beelore.files.wordpress.com/2010/01/thai_beetree.jpg

Some isometry figures from Algorithms textbook.

Fantasy code for 2-3 Tree courtesy of Kevin Wayne