Week 7 - Monday

COMP 2100

Last time

- What did we talk about last time?
- Traversals
- BST delete

Questions?

Project 2

Infix to Postfix Converter

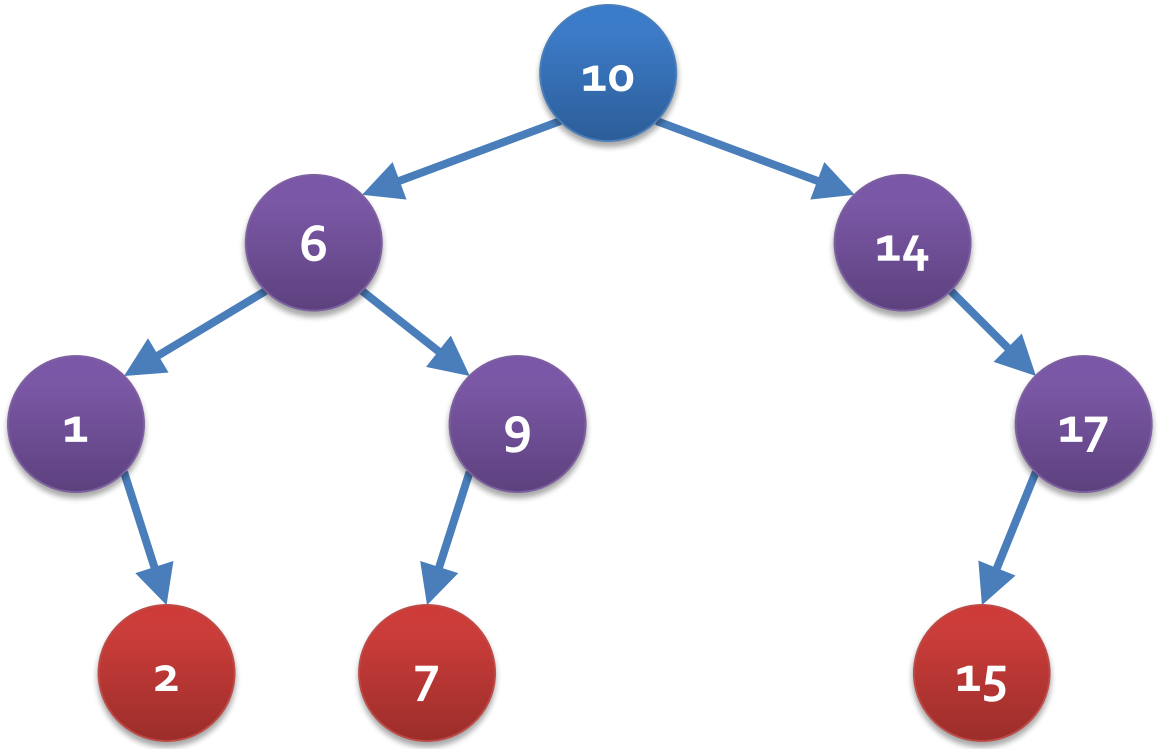
Breadth First Search

What if we didn't have a BST?

- Maybe we still have a binary tree, but we don't have any guarantees about ordering
- How would you search for something?
- We could use preorder or postorder traversals
  - These are types of depth first searches
  - You go to the bottom of the tree before you come back
- What if we thought what we are looking for might be close to the top?

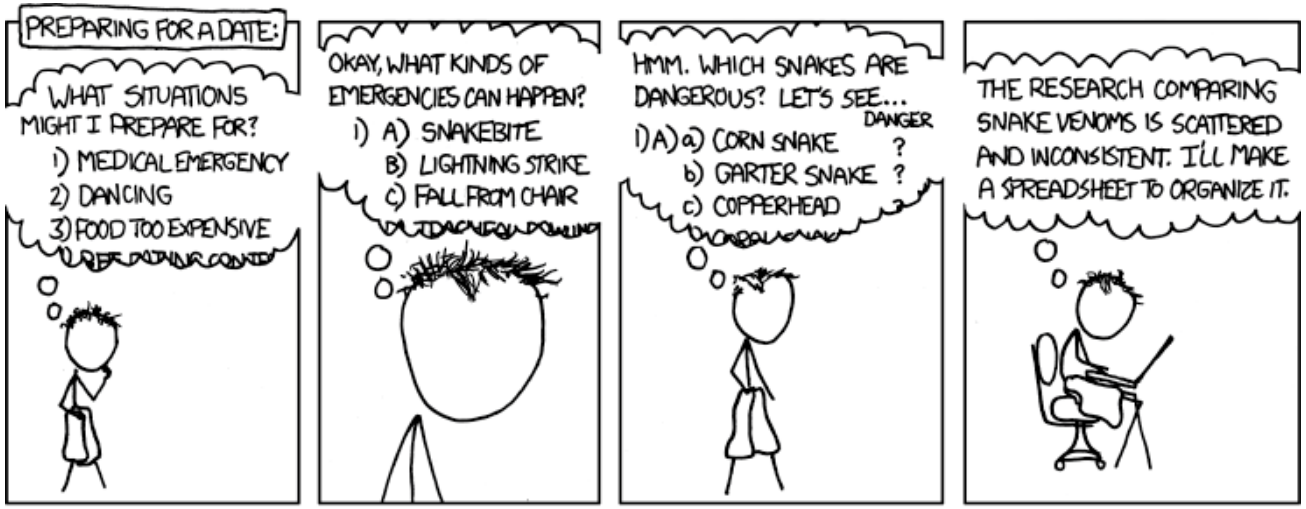
Level order traversal

- The most logical breadth first traversal visits each level of a tree in order:



- 10 6 14 1 9 . 17 . 2 7 . 15 . . . . . .

Breadth first search and dating



From: http://xkcd.com/761/

Breadth first algorithm

- For depth first traversals, we used a stack
- What are we going to use for a BFS?
  - A queue!
- Algorithm:
  1. Put the root of the tree in the queue
  2. As long as the queue is not empty:
     a) Dequeue the first element and process it
     b) Enqueue all of its children

Breadth first implementation

- Write a level order (breadth first) traversal
- Hint: Use an explicit queue
- Non-recursive!

```
public void levelOrder()
```

2-3 Trees

Balance in all things

2-3 trees

- A 2-3 search tree is a data structure that maintains balance
- It is actually a ternary tree, not a binary tree
- A 2-3 tree is one of the following three things:
  - An empty tree (null)
  - A 2-node (like a BST node) with a single key, smaller data on its left and larger values on its right
  - A 3-node with two keys and three links, all key values smaller than the first key on the left, between the two keys in the middle, and larger than the second key on the right

2-3 tree properties

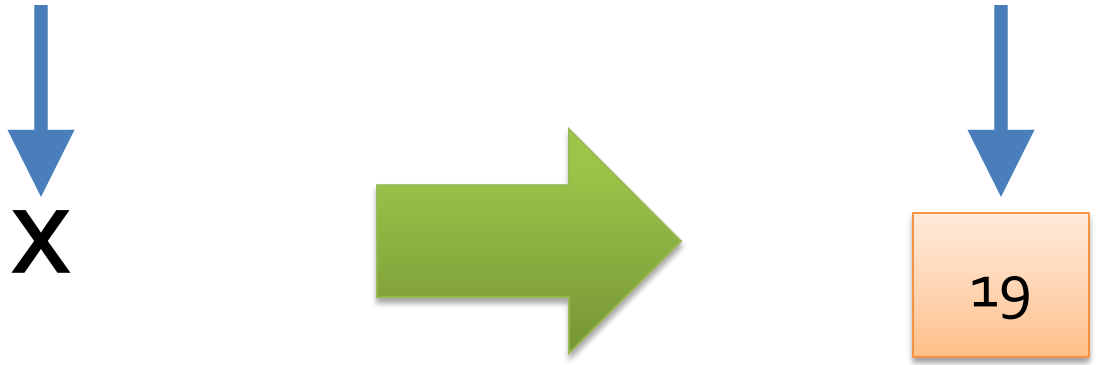
- The key thing that keeps a 2-3 search tree balanced is that all leaves are on the same level
- Only leaves have null links
- Thus, the maximum depth is somewhere between the $\log_3 n$ (the best case, where all nodes are 3-nodes) and $\log_2 n$ (the worst case, where all nodes are 2-nodes)

How does that work?

- We build from the bottom up
- Except for an empty tree, we never put a new node in a null link
- Instead, you can add a new key to a 2-node, turning it into a 3-node
- Adding a new key to a 3-node forces it to break into two 2-nodes

Simplest case: empty tree

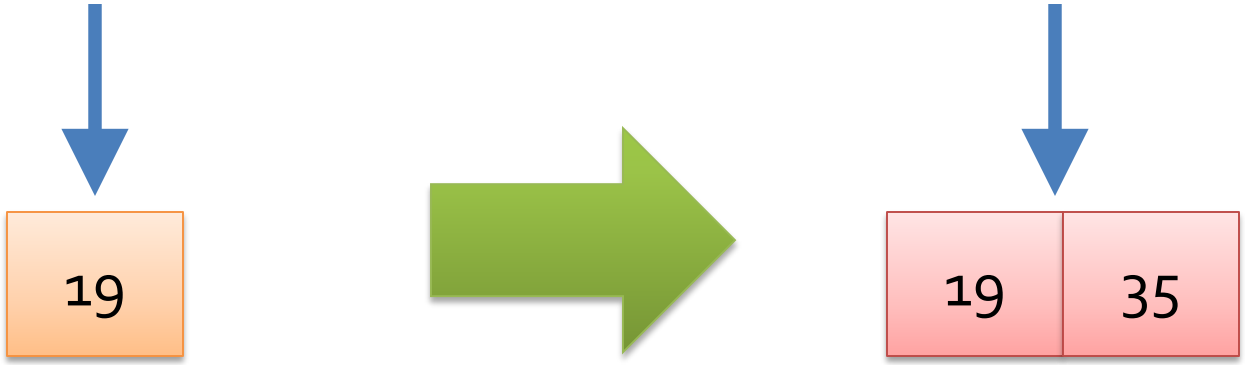
- Starting with an empty tree, we put in a new 2-node:



Add 19

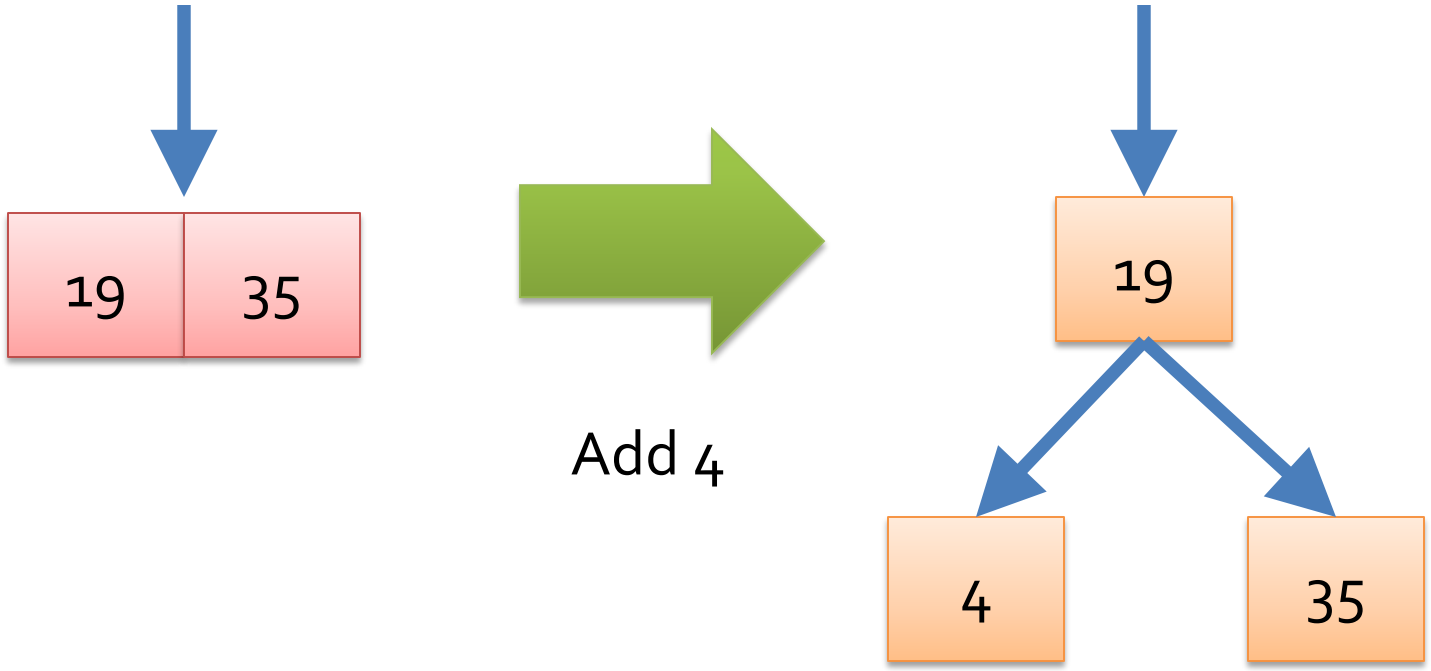
Next case: add to 2-node

- When adding to a 2-node, make it a 3-node, we put in a new 2-node:



Next case: add to 3-node

- When adding to a 3-node, the 3-node breaks into two 2-nodes, bringing the middle value up:



Adding to a 3-node continued

- When breaking a 3-node into two parts, you move the middle value up
- If the node above is empty, it's a new 2-node
- If the node above is a 2-node, it becomes a 3-node
- If the node above is another 3-node, it also breaks into 2-nodes, which might cascade up the tree

2-3 tree practice

- Add the following keys to a 2-3 tree:
  - 62
  - 11
  - 32
  - 7
  - 45
  - 24
  - 88
  - 25
  - 28
  - 90

2-3 tree running times

- Because of the guarantees about the depth of the tree, we the following running times for 2-3 search trees
  - $\Theta(\log n)$ insert
  - $\Theta(\log n)$ delete (messy, but true)
  - $\Theta(\log n)$ find (not that different from a BST find)

2-3 tree implementation

- How do we implement a 2-3 tree?
- Answer: We don't.
- It is (of course) possible, but it involves having weird 2-nodes and 3-nodes that inherit from some abstract node class or interface
- It's a huge pain
  - Note that 2-3 trees are essentially a special case of a B-tree, and someone does have to implement those
- Instead, we use red-black trees which are structurally the same as 2-3 trees (if you squint)

Red-black trees

An interlude in a formicary

- One hundred ants are walking along a meter long stick
- Each ant is traveling either to the left or the right with a constant speed of 1 meter per minute
- When two ants meet, they bounce off each other and reverse direction
- When an ant reaches an end of the stick, it falls off
- Will all the ants fall off?
- What is the longest amount of time that you would need to wait to guarantee that all ants have fallen off?

"As if"

- On the previous slide, we can look at the problem "as if" ants were passing through each other with no effect
- This idea of looking at a problem as if it is something else can make solving it easier
- Coding up a 2-3 tree is annoying (but possible)
- By creating a data structure that is (somehow) equivalent, we can get the job done in an easier way
- Sometimes the way we implement an algorithm and the way we analyze it are different

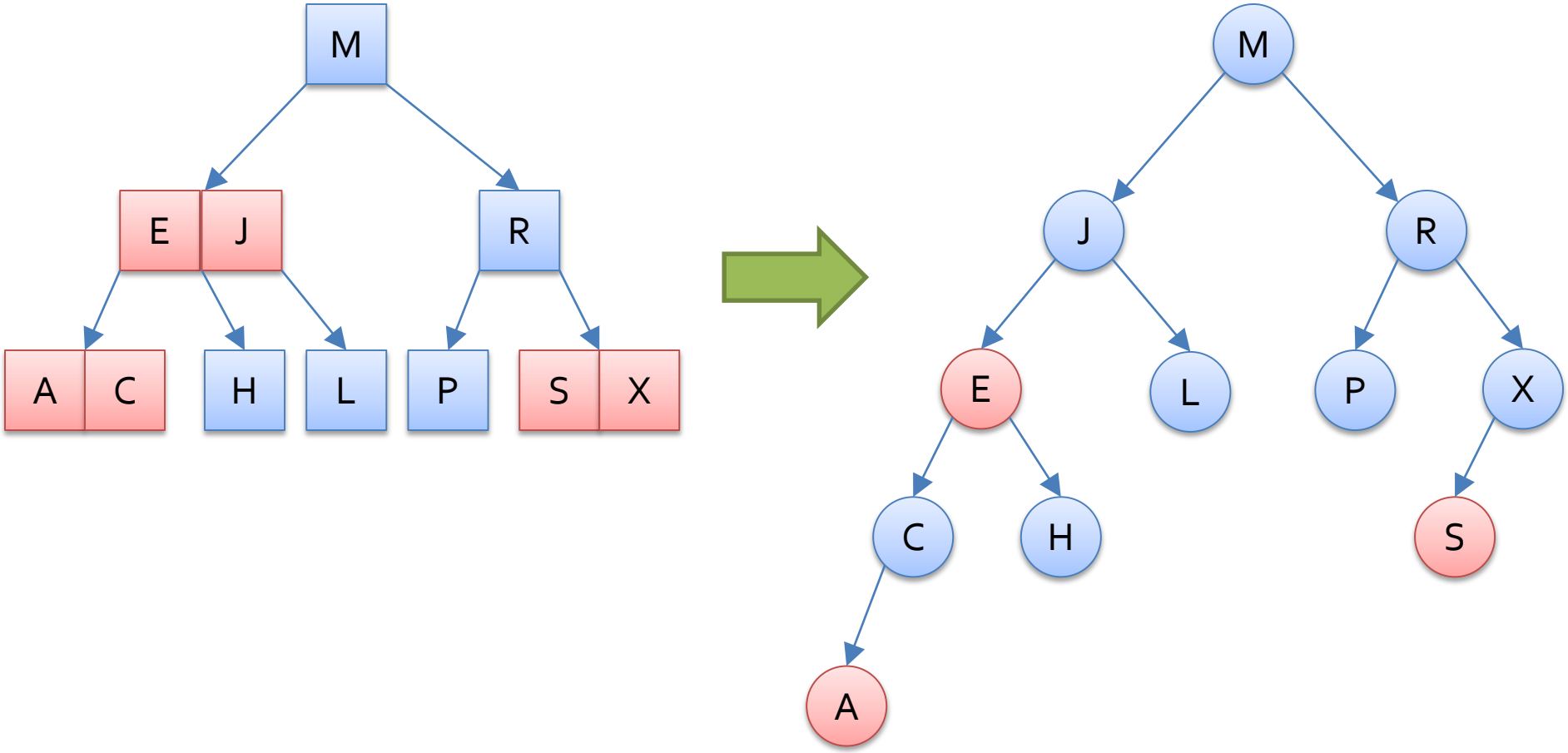
Red-black trees

- A red-black tree is a form of binary search tree
- Each node looks like a regular BST node, with one additional piece of information: color
  - A node can either be **red** or **black**
  - Null values are considered black
- The color allows us to simulate a 2-3 tree
  - We can think of a red node is actually part of a 3 node with its parent

Red-black tree definition

- A red-black tree is a BST with red and black nodes and the following properties:
  - Red nodes lean left from their parents
  - No node has two red children
  - The tree has **perfect black balance**
    - In other words, every path from the root to a null has the same number of black nodes on the way
    - The length of this path is called the **black height** of the tree
- The book describes the link as having a color (which is probably easier to think about), but the color has to be stored in the node

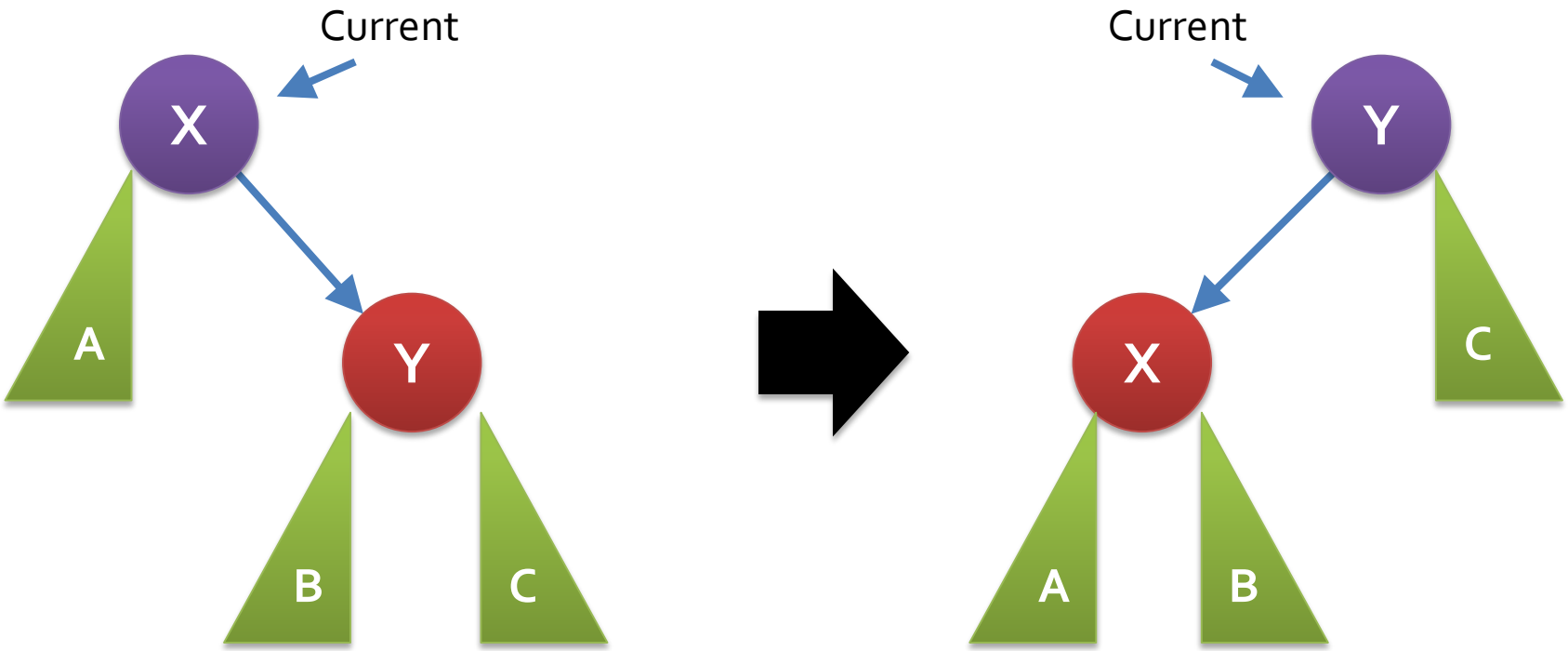
2-3 tree mapping to a red-black tree



Building the tree

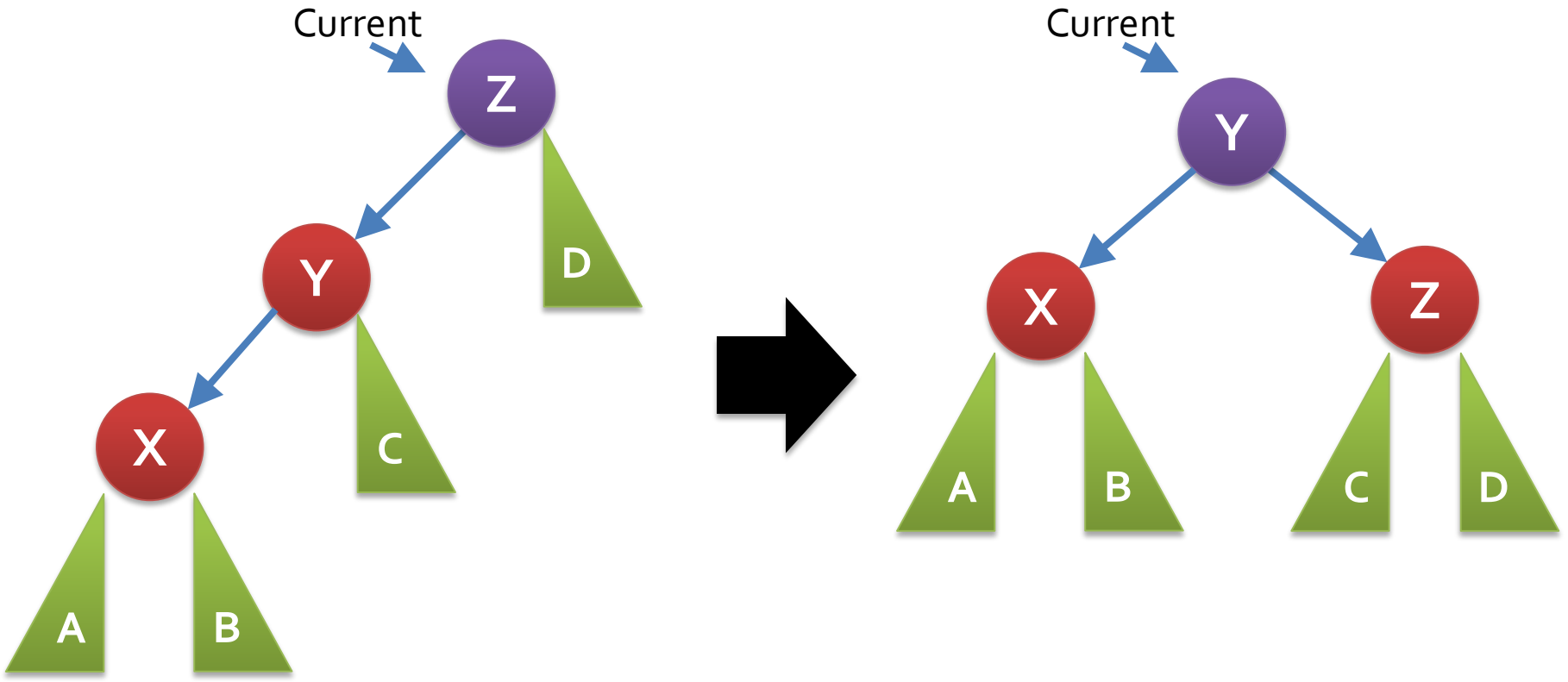
- We can do an insertion with a red-black tree using a series of rotations and recolors
- We do a regular BST insert
- Then, we work back up the tree as the recursion unwinds
  - If the right child is red, we rotate the current node left
  - If the left child is red and the left child of the left child is red, we rotate the current node right
  - If both children are red, we recolor them black and the current node red
- **You have to do all these checks, in order!**
  - Multiple rotations can happen
- It doesn't make sense to have a red root, so we always color the root black after the insert

Left rotation



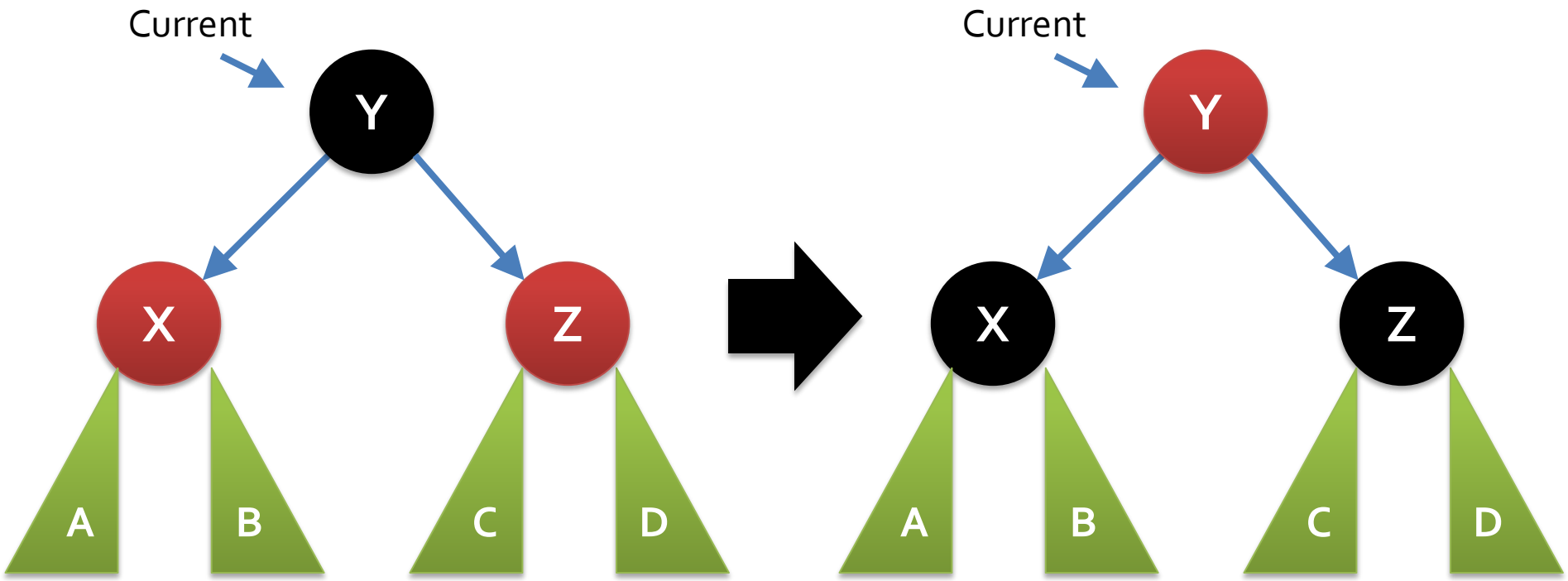
We perform a left rotation
when the right child is red

Right rotation



We perform a right rotation when the left child is red and its left child is red

Recolor



We recolor both children and the current node when both children are red

Red-black tree practice

- Add the following keys to a red-black tree:
  - 62
  - 11
  - 32
  - 7
  - 45
  - 24
  - 88
  - 25
  - 28
  - 90
- Hint: Add to a 2-3 tree, then convert to red-black

Analysis of red-black trees

- The height of a red-black tree is no more than 2 log ***n***
- Find is Θ(*height*), so find is Θ(log ***n***)
- Since we only have to go down that path and back up to insert, insert is Θ(log ***n***)
- Delete in red-black trees is messy, but it is also actually Θ(log ***n***)

Upcoming

Next time…

- Finish red-black trees
- AVL trees
- Balancing trees by construction

Reminders

- Keep working on Project 2
  - Due Friday!
- Keep reading Section 3.3