Week 11 - Friday

COMP 2100

Last time

- What did we talk about last time?
- More on hard problems
- P vs. NP
- Started review

Questions?

Project 3

Assignment 6

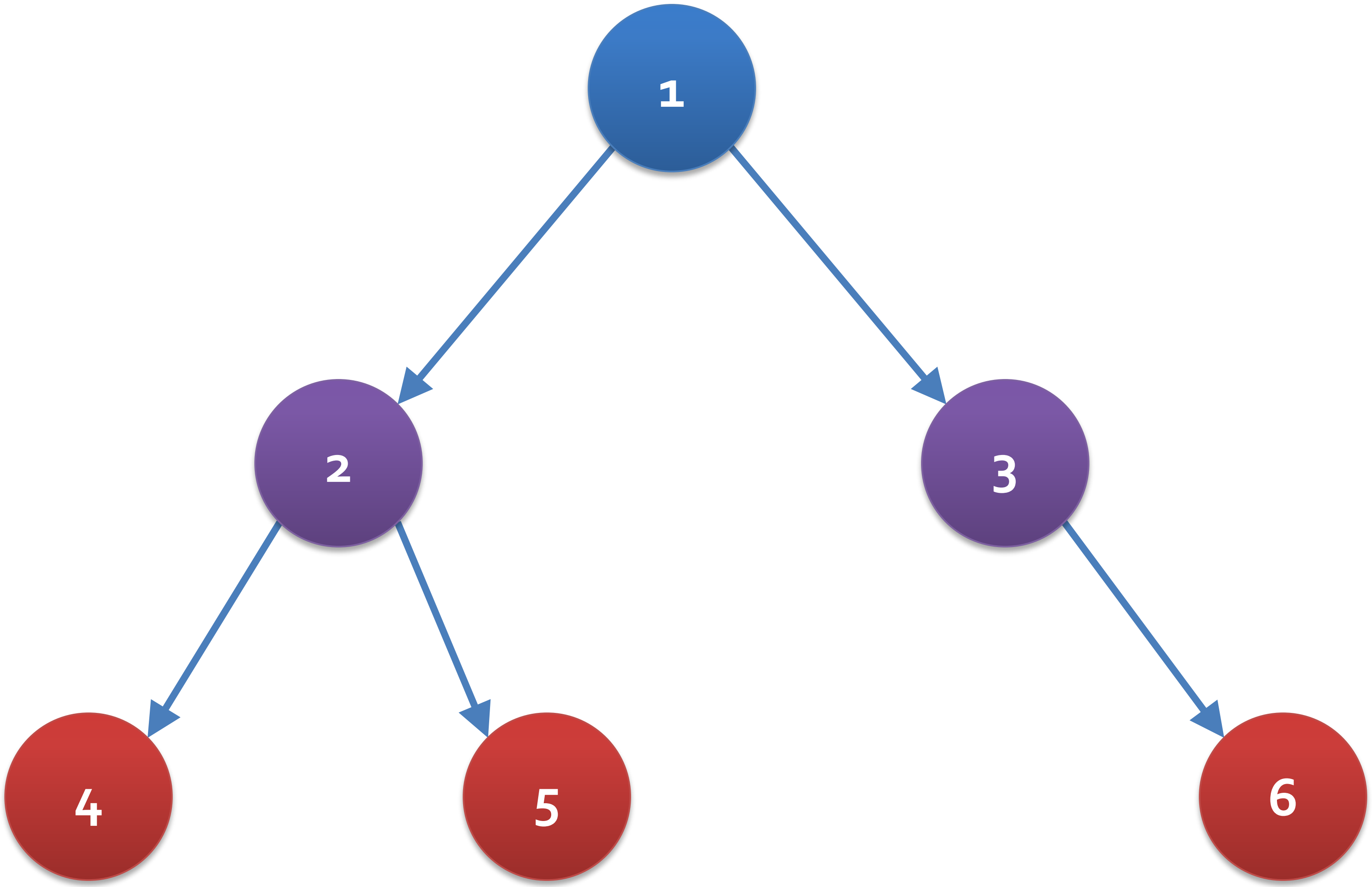
Review

Trees

Binary tree

- A binary tree is a tree such that each node has two or fewer children
- The two children of a node are generally called the **left child** and the **right child**, respectively

Binary tree



Binary tree terminology

- **Full binary tree:** every node other than the leaves has two children
- **Perfect binary tree:** a full binary tree where all leaves are at the same depth
- **Complete binary tree:** every level, except possibly the last, is completely filled, with all nodes to the left
- **Balanced binary tree:** the depths of all the leaves differ by at most 1

Binary search tree (BST)

- A binary search tree is binary tree with three properties:
  1. The left subtree of the root only contains nodes with keys less than the root's key
  2. The right subtree of the root only contains nodes with keys greater than the root's key
  3. Both the left and the right subtrees are also binary search trees

Purpose of a BST

- Keeping data organized
  - Easy to produce a sorted order in O(***n***) time

- Find, add, and delete are all O(log ***n***) time if the tree is balanced

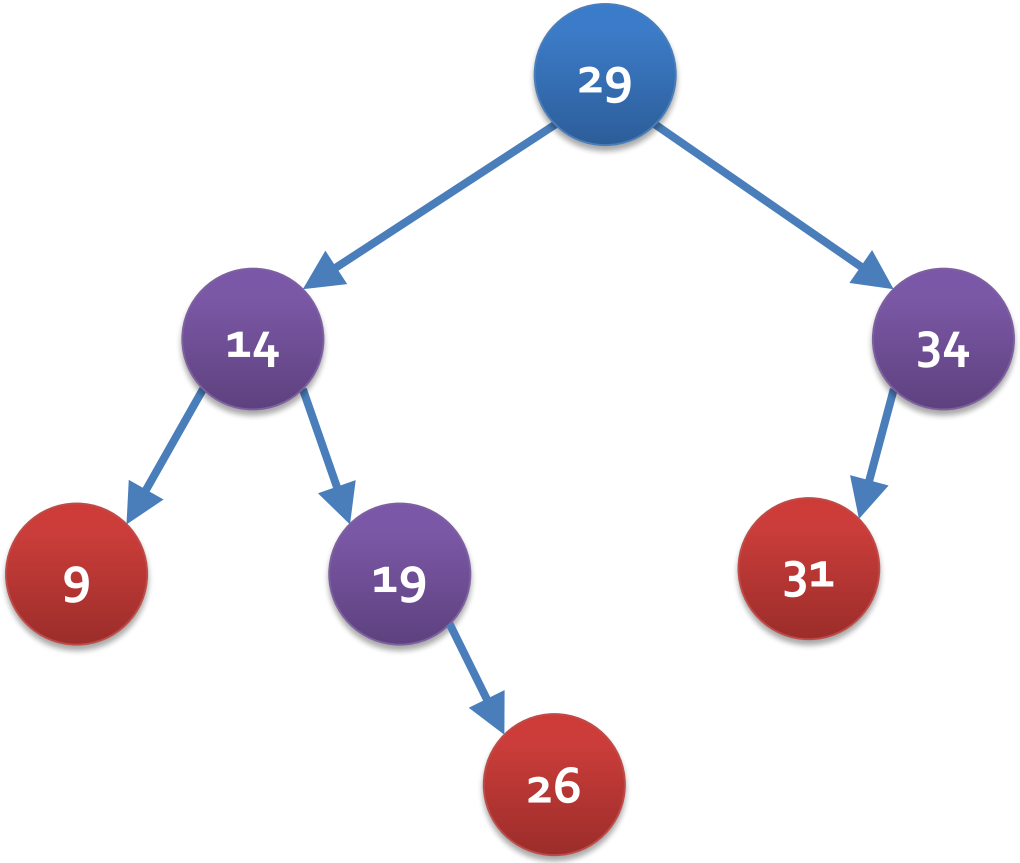
Basic BST class

```
public class Tree {
  private static class Node {
    public int key;
    public String value;
    public Node left;
    public Node right;
  }

  private Node root = null;

  …
}
```

Traversals

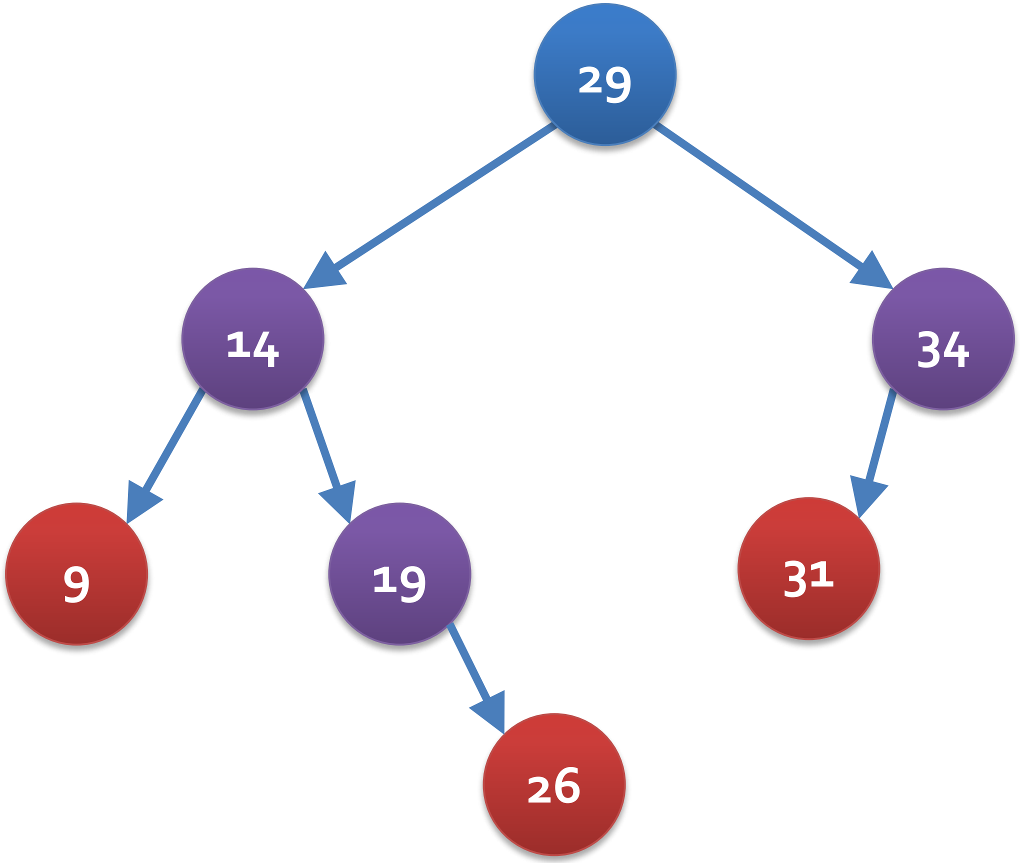
- Visiting every node in a tree is called a **traversal**
- There are four traversals we are interested in:
  - Preorder (NLR)
  - Postorder (LRN)
  - Inorder (LNR)
  - Level Order
- The first three are depth first, the last is breadth first

Preorder



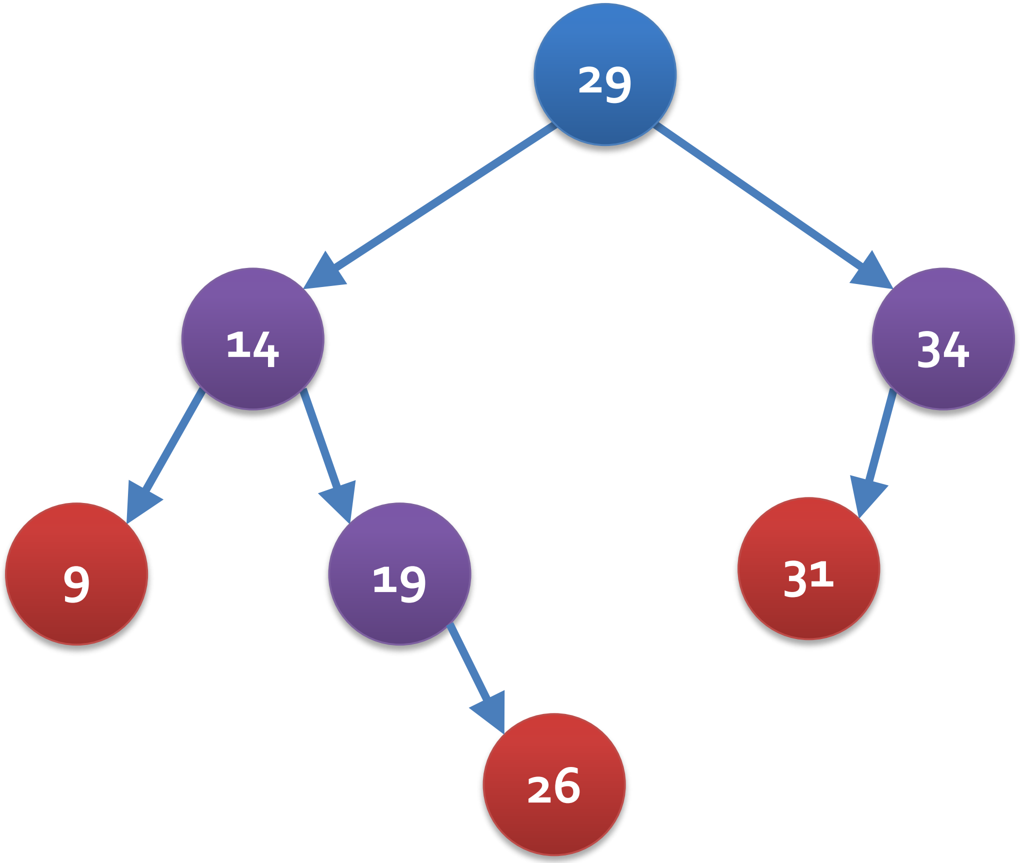
29 14 9 . . 19 . 26 . . 34 31 . . .

Postorder



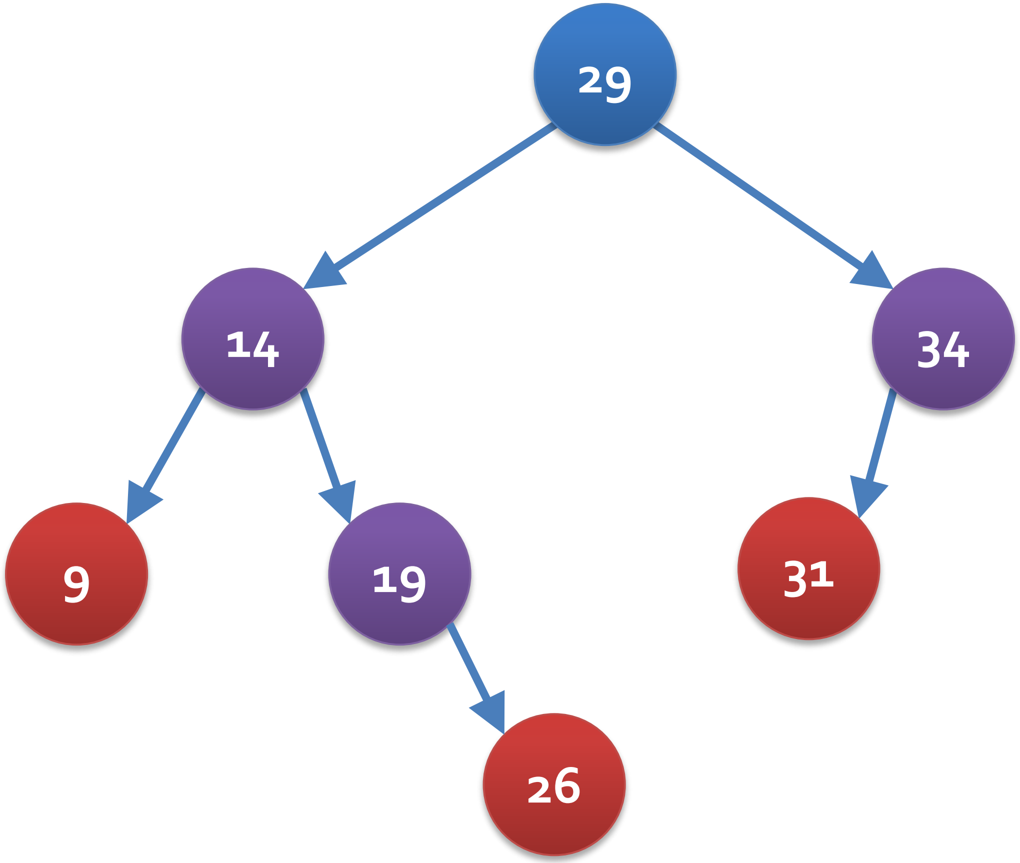
. . 9 . . . 26 19 14 . . 31 . 34 29

Inorder



. 9 . 14 . 19 . 26 . 29 . 31 . 34 .

Level order



29 14 34 9 19 31 . . . . 26 . . . .

Level order 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

Balanced Trees

Balancing trees

- We can have a balanced tree by:
  - Doing red-black (or AVL) inserts
  - Balancing a tree by construction (sort, then add)
  - DSW algorithm: completely unbalance then rebalance

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)

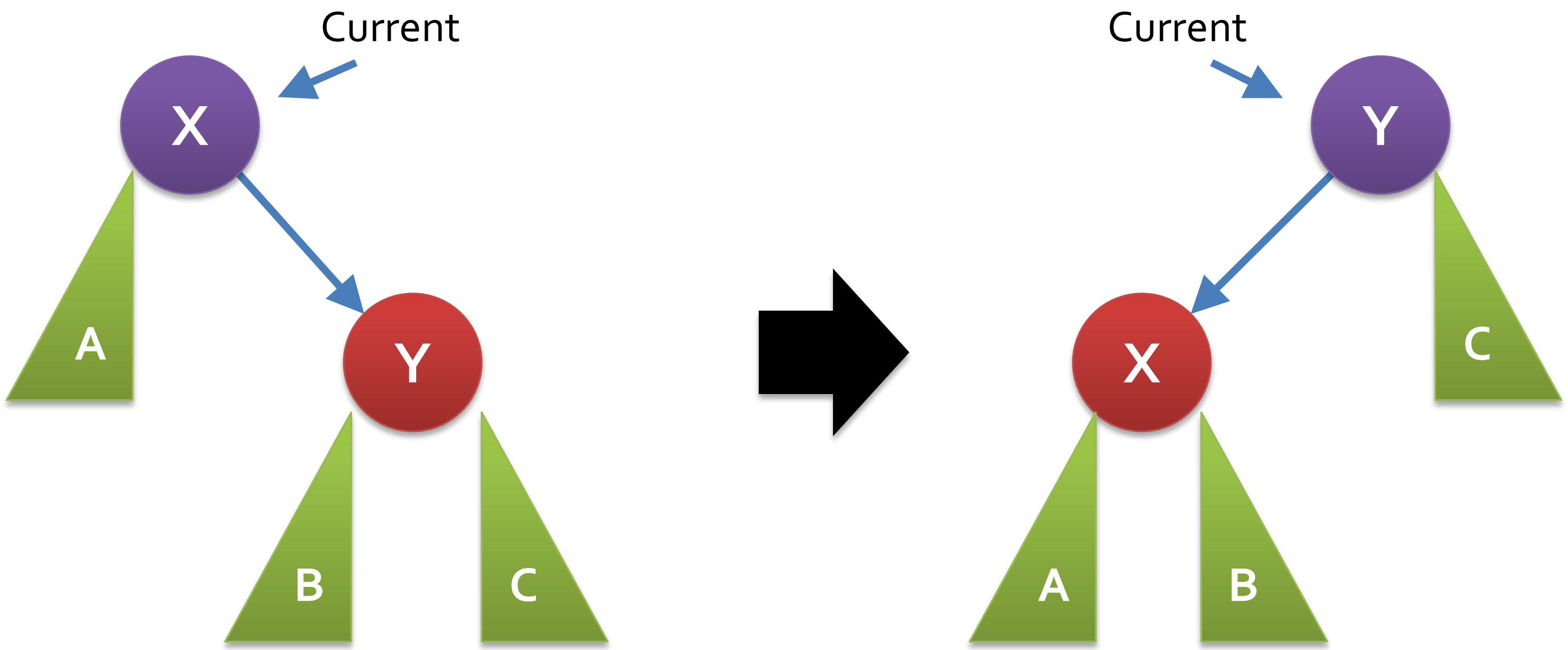
How does that work?

- 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

Building a red-black tree

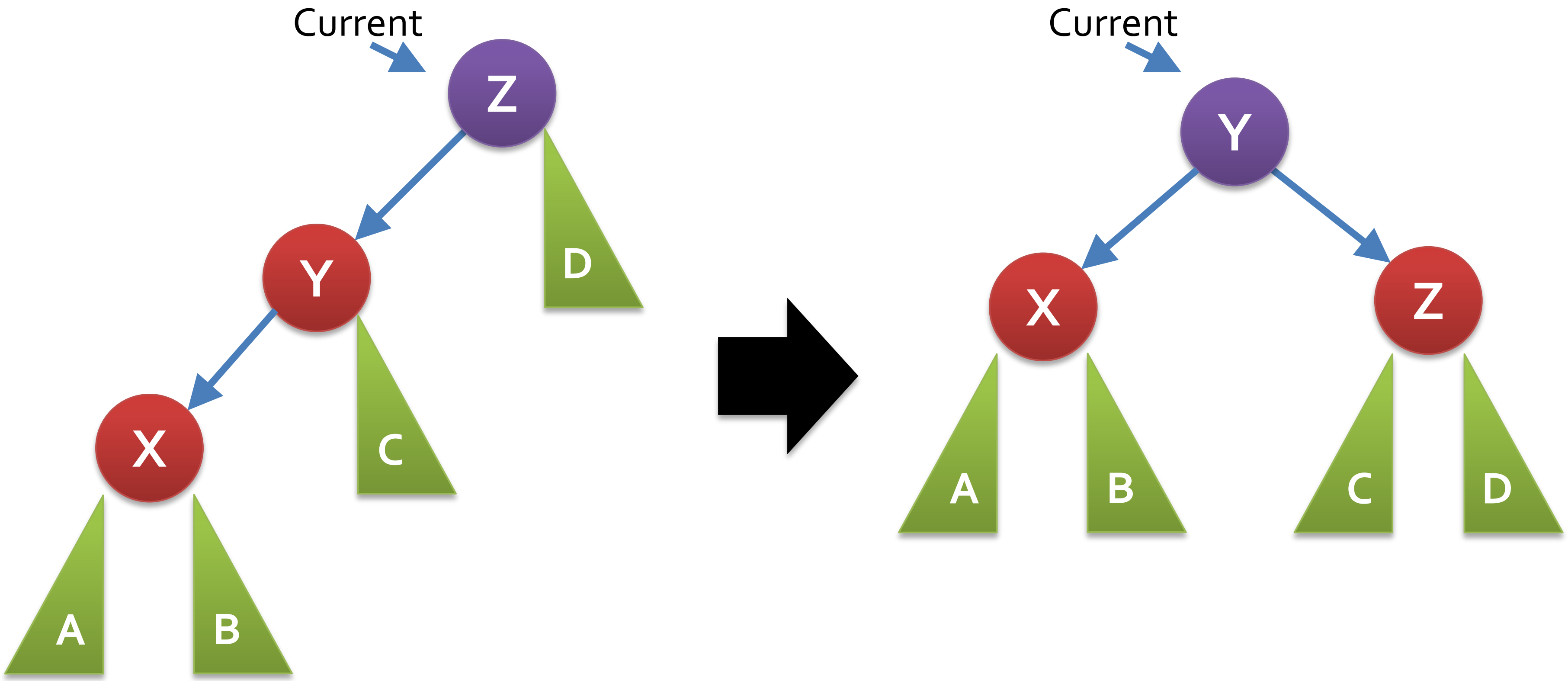
- We can do an insertion with a red-black tree using a series of rotations and recolors
- We do a regular BST insert
- Then, we work back up the tree as the recursion unwinds
  - If the right child is red and the left is black, we rotate the current node left
  - If the left child is red and the left 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



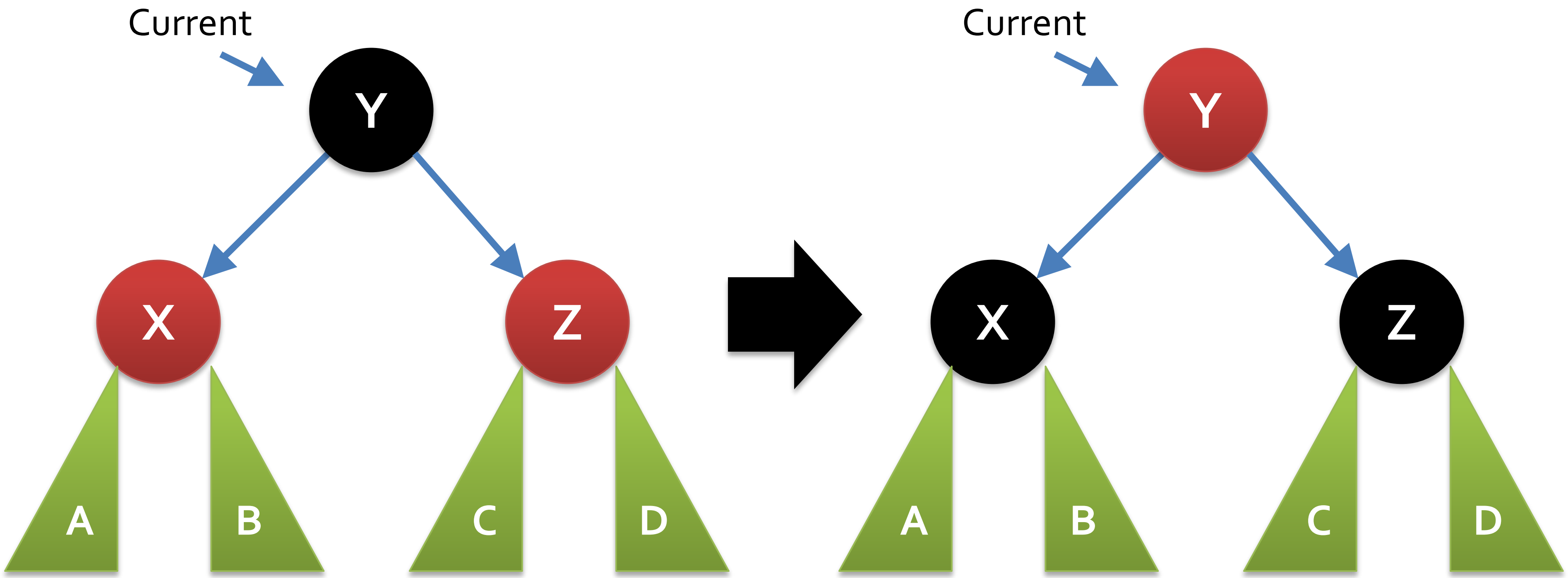
We perform a left rotation
when the right child is red

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

Exam hints

- Learn how to do 2-3 tree insertions really well
- Then, learn how you can map a 2-3 tree onto a red-black tree
- It's much easier to make a 2-3 tree and then figure out the corresponding red-black tree than it is to build a red-black tree from scratch

Hash Tables

Hash tables: theory

- We make a huge array, so big that we'll have more spaces in the array than we expect data values
- We use a **hashing function** that maps items to indexes in the array
- Using the hashing function, we know where to put each item but also where to look for a particular item

Hash table: issues

- We are using a hash table for a space/time tradeoff
- Lots of space means we can get down to O(1)
- How much space do we need?
  - When the table gets too full, we may need to rehash everything
- How do we pick a good hashing function?
- What happens if two values **collide** (map to the same location)

Collisions

- With open addressing, we look for some empty spot in the hash table to put the item
- There are a few common strategies
  - Linear probing
  - Quadratic probing
  - Double hashing
- Alternatively, we can use chaining

Graphs

Graphs

- Edges
- Nodes
- Types
  - Undirected
  - Directed
  - Multigraphs
  - Weighted
  - Colored
  - Triangle inequality

Traversals

- Depth First Search
  - Cycle detection
  - Connectivity
- Breadth First Search

Dijkstra's Algorithm

- Start with two sets, ***S*** and ***V***:
  - ***V*** has everything in it

1. Set the distance to the starting node to 0
2. Set the distance to all other nodes to ∞
3. Find the node ***u*** in ***V*** with the smallest ***d***(***u***)
4. For every neighbor ***v*** of ***u*** in V
   a) If ***d***(***v***) > ***d***(***u***) + ***d***(***u***,***v***)
   b) Set ***d***(***v***) = ***d***(***u***) + ***d***(***u***,***v***)
5. Move ***u*** from ***V*** to ***S***
6. If ***V*** is not empty, go back to Step 2

Minimum Spanning Tree (MST)

- Start with two sets, ***S*** and ***V***:
  - ***S*** has the starting node in it
  - ***V*** has everything else

1. Find the node ***u*** in ***V*** that is closest to any node in ***S***
2. Put the edge to ***u*** into the MST
3. Move ***u*** from ***V*** to ***S***
4. If ***V*** is not empty, go back to Step 1

Note

- Exam 2 will **not** cover:
  - Euler paths and cycles
  - Bipartite graphs and matching
  - Network flow
  - Hamiltonian cycles or TSP
  - NP-completeness

Basic BST class

```
public class Tree {
  private static class Node {
    public int key;
    public Object value;
    public Node left;
    public Node right;
  }

  private Node root = null;

  …
}
```

Sample programming question

- Write a method that counts the number of nodes in a BST, using the definition of a BST on a previous slide

```
private static int count(Node node)
```

Proxy:

```
public int count() {
  return count( root );
}
```

Sample programming question

- Write a method that tests to see if a binary tree maintains its ordering property (left is smaller, right is larger)

```
private static boolean isSearchTree(Node node, int
  min, int max)
```

Proxy:

```
  public boolean isSearchTree() {
    return isSearchTree(root, Integer.MIN_VALUE,
    Integer.MAX_VALUE);
  }
```

Sample programming question

- Write a method to determine if two **`String`** objects are permutations of each other
  - Do it in $\Theta(n)$ time where n is the length of the **`String`** values
  - Hint: Use a hash table: **`HashMap<Character,Integer>`**

Answers to Sample Programming Questions

Warning!

It is unwise to look at these solutions until you have attempted them on your own.

Understanding code is a different skill from creating code.

Answer to programming question 1

```
private static int count(Node node) {
    if( node == null )
        return 0;
    else
        return 1 + count(node.left) +
            count(node.right);
}
```

Answer to programming question 2

```
private static boolean isSearchTree(Node node, int
  min, int max) {
     if( node == null )
          return true;
     else
          return node.key > min &&
          node.key < max &&
          isSearchTree(node.left, min, node.key) &&
          isSearchTree(node.right, node.key, max);
}
```

Answer to programming question 3

```
private static boolean arePermutations(String s1, String s2) {
        if (s1.length() != s2.length())
                return false;
        HashMap<Character,Integer> map = new HashMap<Character,Integer>();
        for (int i = 0; i < s1.length(); ++i) {
                char c = s1.charAt(i);
                if( map.containsKey(c) )
                        map.put(c, map.get(c) + 1);
                else
                        map.put(c, 1);
        }
        for (int i = 0; i < s2.length(); ++i) {
                char c = s2.charAt(i);
                if( map.containsKey(c) ) {
                        int count = map.get(c);
                        if( count == 0 )
                                return false;
                        else
                                map.put(c, count - 1);
                }
                else
                        return false;
        }
        return true;
}
```

Upcoming

Next time…

- Exam 2!

Reminders

- Finish Project 3
  - **Due tonight by midnight!**
- Review chapters 3 and 4 for Exam 2
  - **Next Monday!**