Week 16 - Monday

COMP 2100

Last time

- What did we talk about last time?
- Review up to Exam 1

Questions?

Project 4

Student Questions

Recursion

Recursion

- Base case
  - Tells recursion when to stop
  - Can have multiple base cases
  - Have to have at least one or the recursion will never end
- Recursive case
  - Tells recursion how to proceed one more step
  - Necessary to make recursion able to progress
  - Multiple recursive cases are possible

Recursive function example

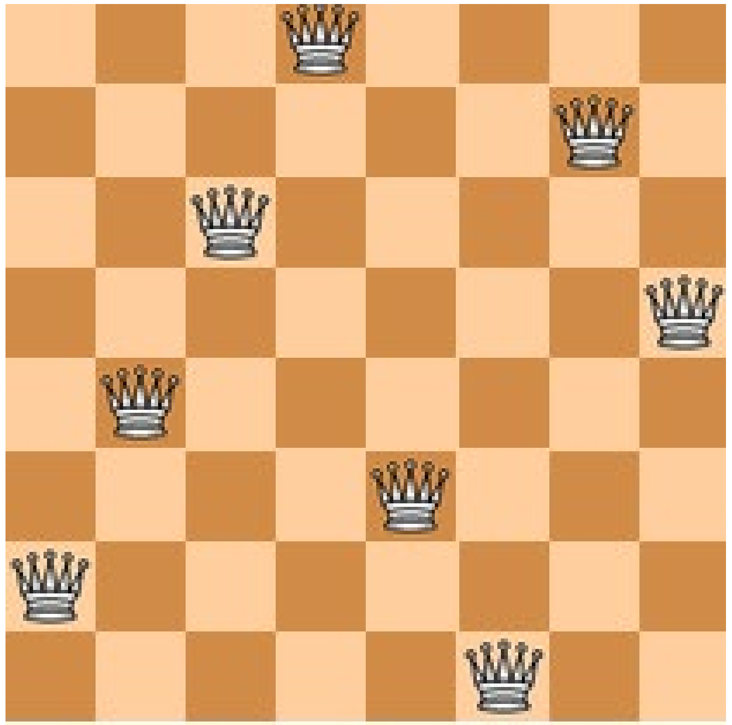
- Factorial:

```
public static int factorial(int n){
  if( n == 1 ) {
    return 1;
  } else {
    return n * factorial( n - 1);
  }
}
```

Recursion Problem

N-Queens

- Given an ***N*** x ***N*** chess board, where ***N*** ≥ 4 it is possible to place ***N*** queens on the board so that none of them are able to attack each other in a given move
- Write a method that, given a value of ***N***, will return the total number of ways that the ***N*** queens can be placed



Symbol tables

- A symbol table goes by many names:
  - Map
  - Lookup table
  - Dictionary
- The idea is a table that has a two columns, a key and a value
- You can store, lookup, and change the value based on the key

Symbol table ADT

- We can define a symbol table ADT with a few essential operations:
  - put(Key key, Value value)
    - Put the key-value pair into the table
  - get(Key key):
    - Retrieve the value associated with key
  - delete(Key key)
    - Remove the value associated with key
  - contains(Key key)
    - See if the table contains a key
  - isEmpty()
  - size()
- It's also useful to be able to iterate over all keys

Trees

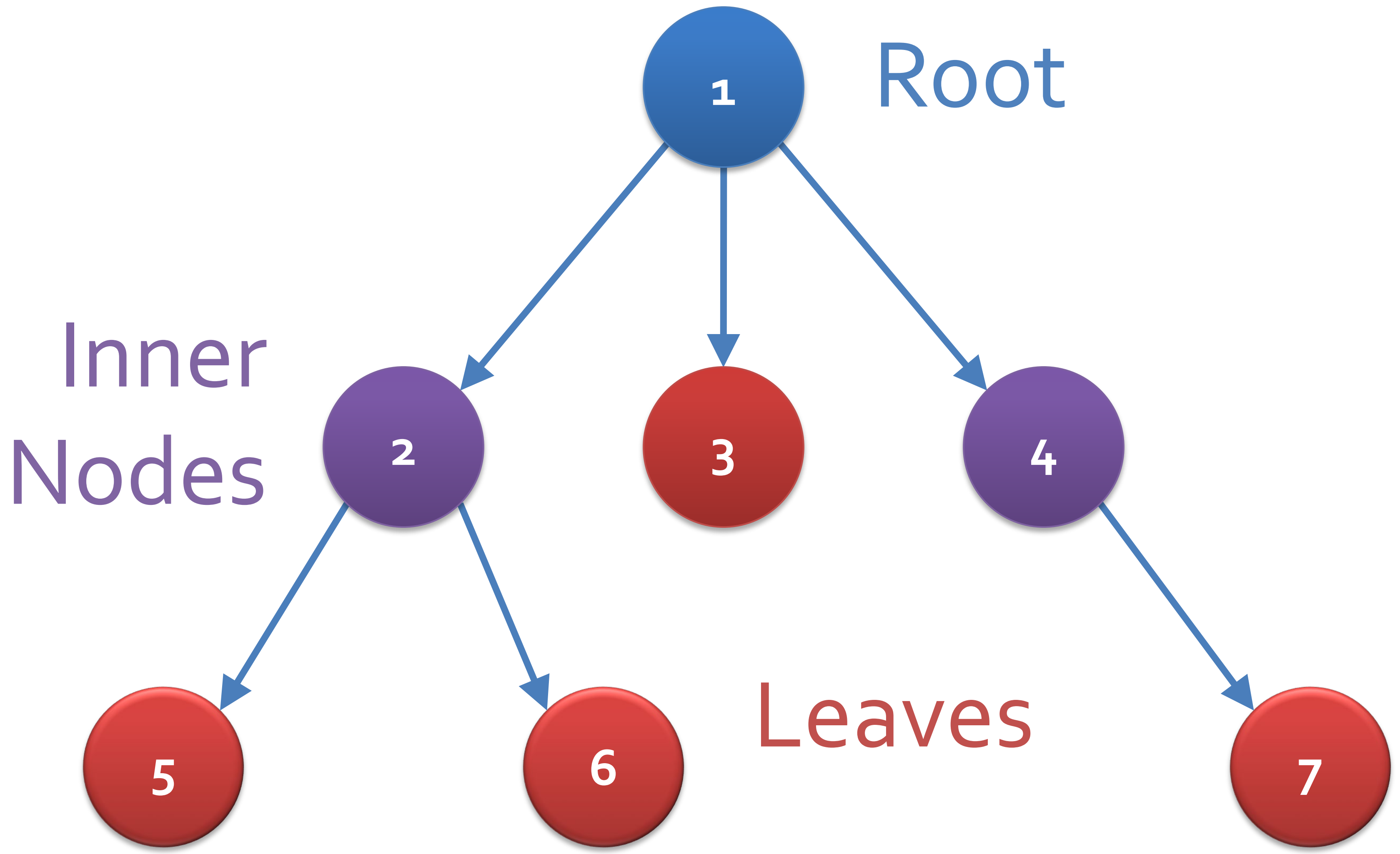
Trees

- A tree is a data structure built out of nodes with children
- A general tree node can have any non-negative number of children
- Every child has exactly one parent node
- There are no loops in a tree
- A tree expressions a hierarchy or a similar relationship

Terminology

- The **root** is the top of the tree, the node which has no parents
- A **leaf** of a tree is a node that has no children
- An **inner node** is a node that does have children
- An **edge** or a **link** connects a node to its children
- The **depth** of a node is the length of the path from a node to its root
- The **height** of the tree is the greatest depth of any node
- A **subtree** is a node in a tree and all of its children
- **Level:** the set of all nodes at a given depth from the root

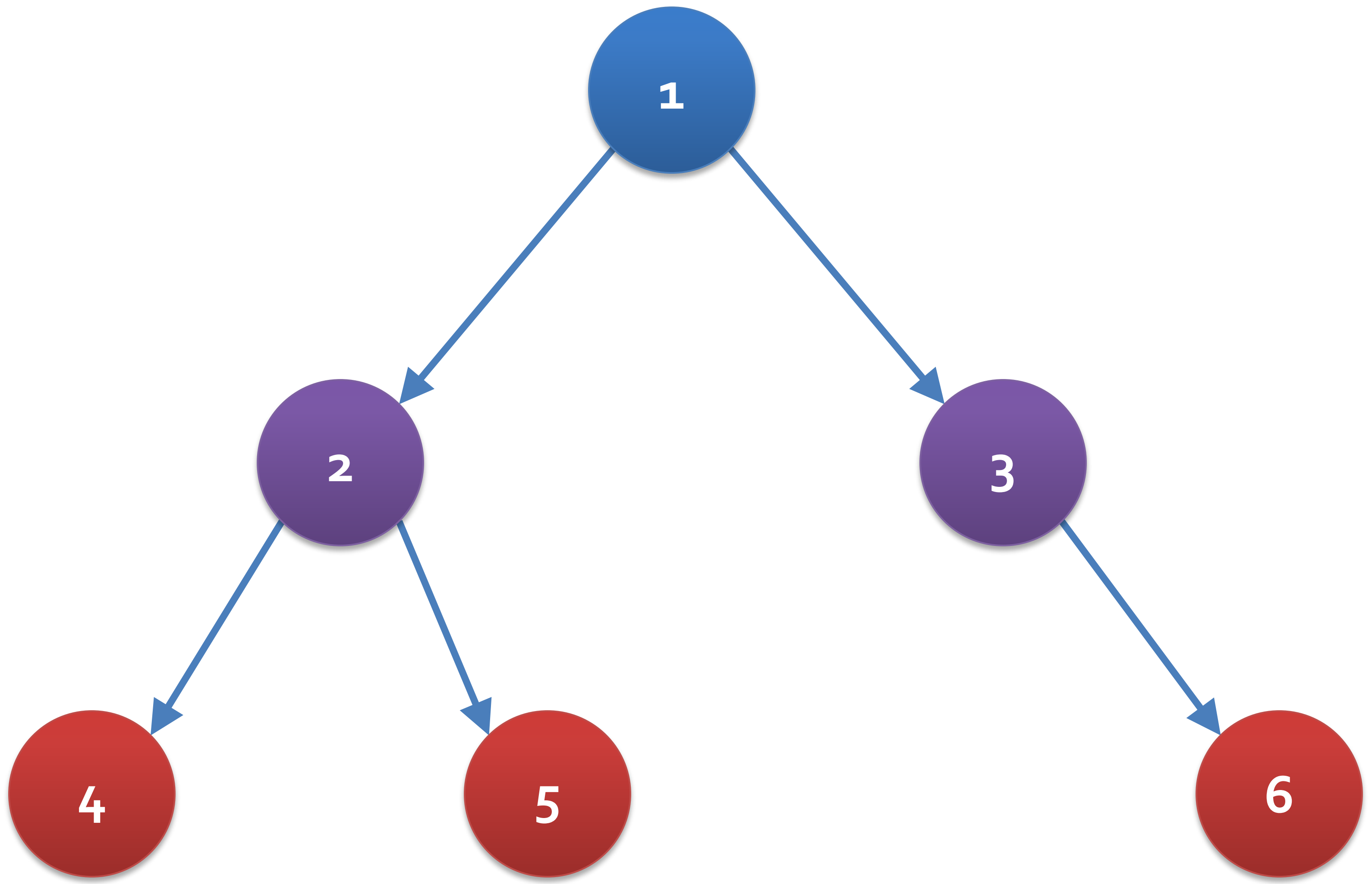
A tree



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

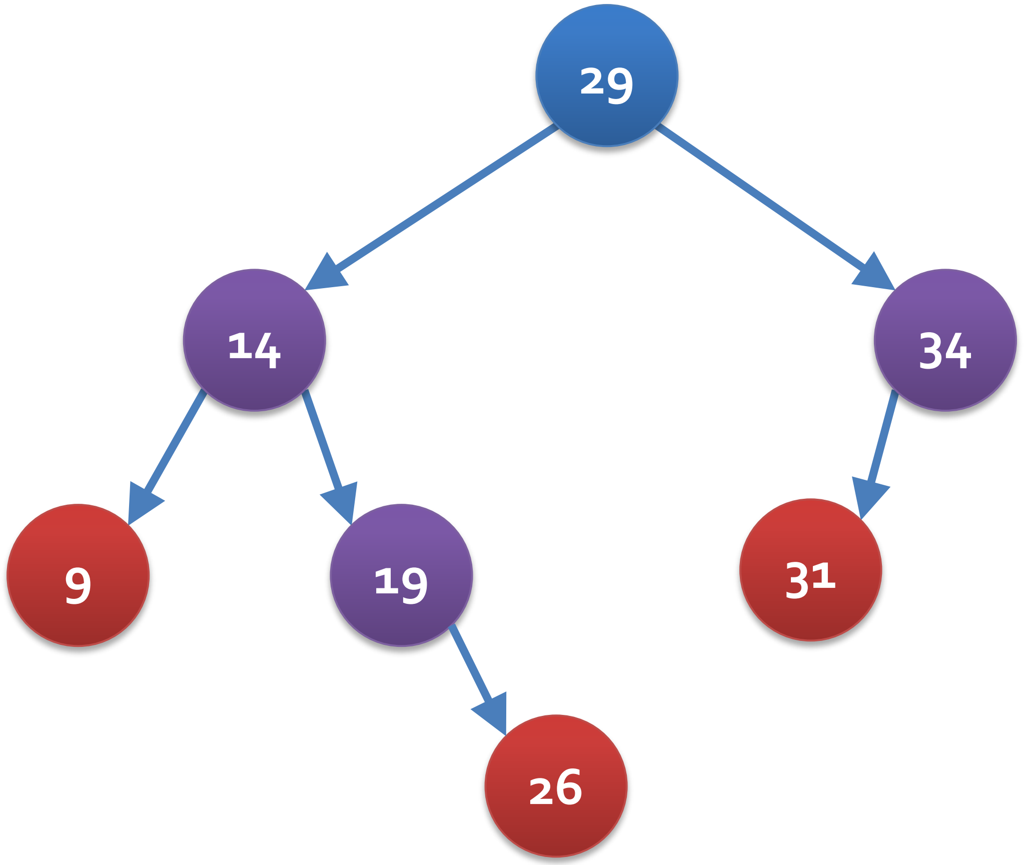
Purpose of a BST

- Keeping data organized
  - Easy to produce a sorted order in O(***n***) time
- Find, add, and delete are all O(log ***n***) time

Basic BST class

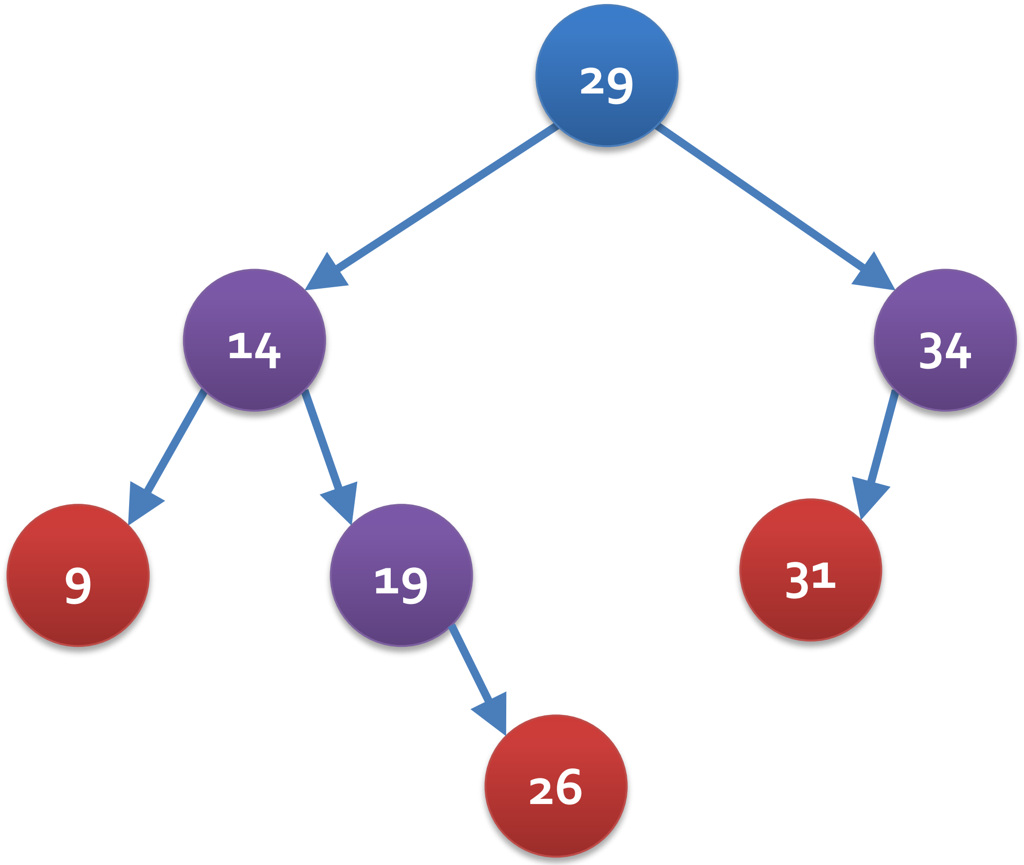
```
public class Tree {
  private static class Node {
    public int key;
    public String value;
    public Node left;
    public Node right;
  }

  private Node root = null;

  …
}
```

Preorder



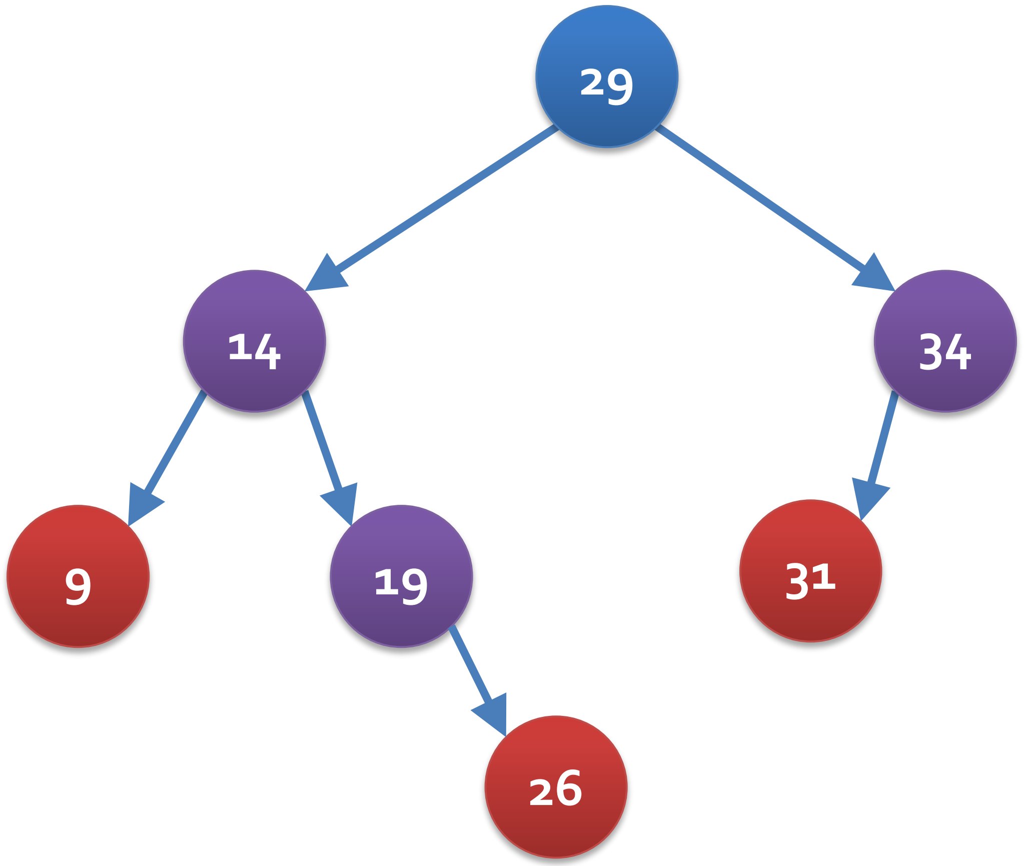
29 14 9 . . 19 . 26 . . 34 31 . . .

Postorder



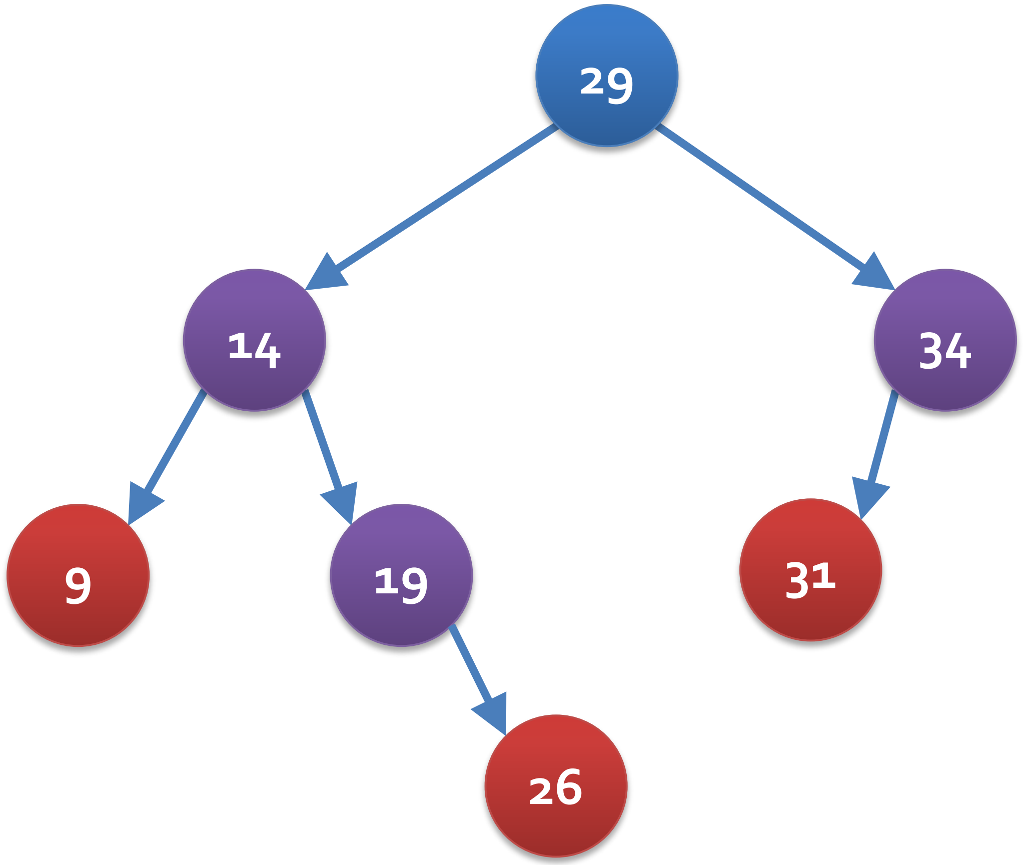
. . 9 . . . 26 19 14 . . 31 . 34 29

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

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?

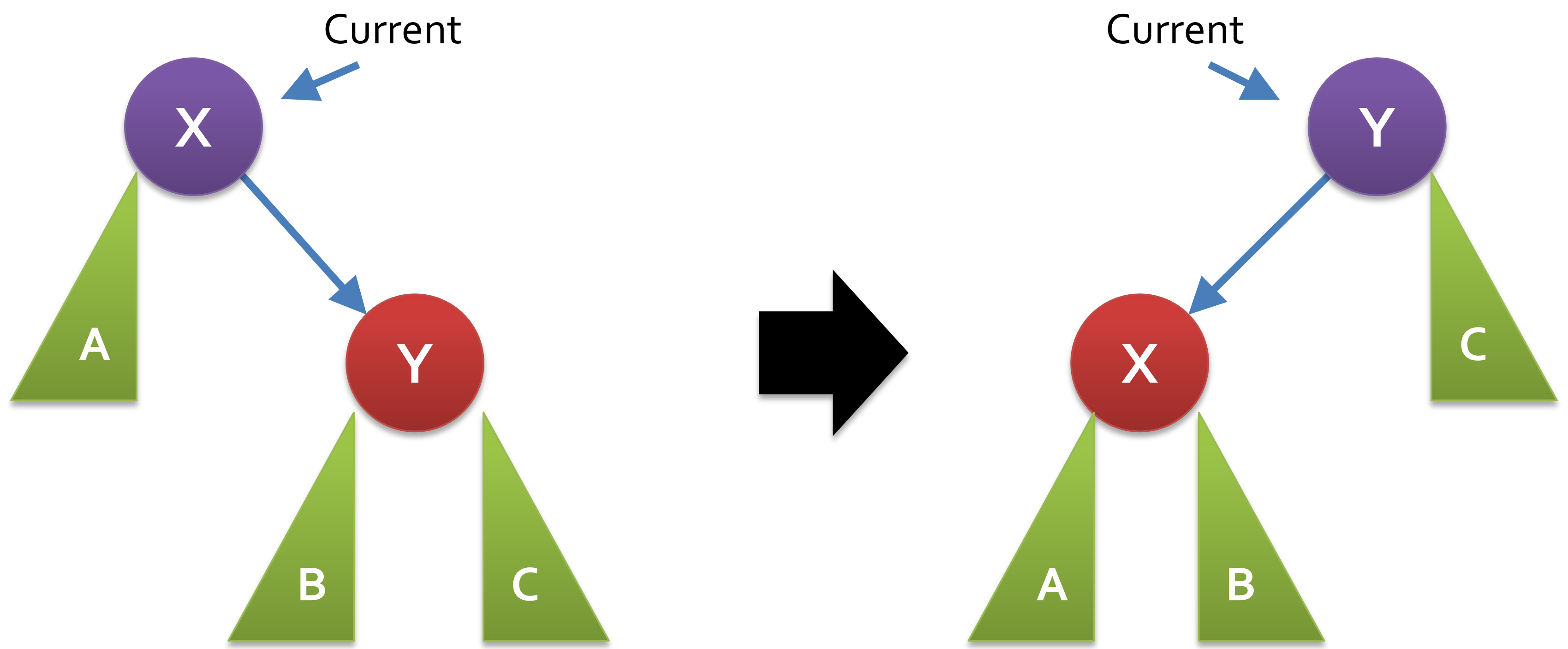
- We build from the bottom up
- Except for an empty tree, we never put a new node in a null link
- Instead, you can add a new key to a 2-node, turning it into a 3-node
- Adding a new key to a 3-node forces it to break into two 2-nodes

Red-black Trees

Building the tree

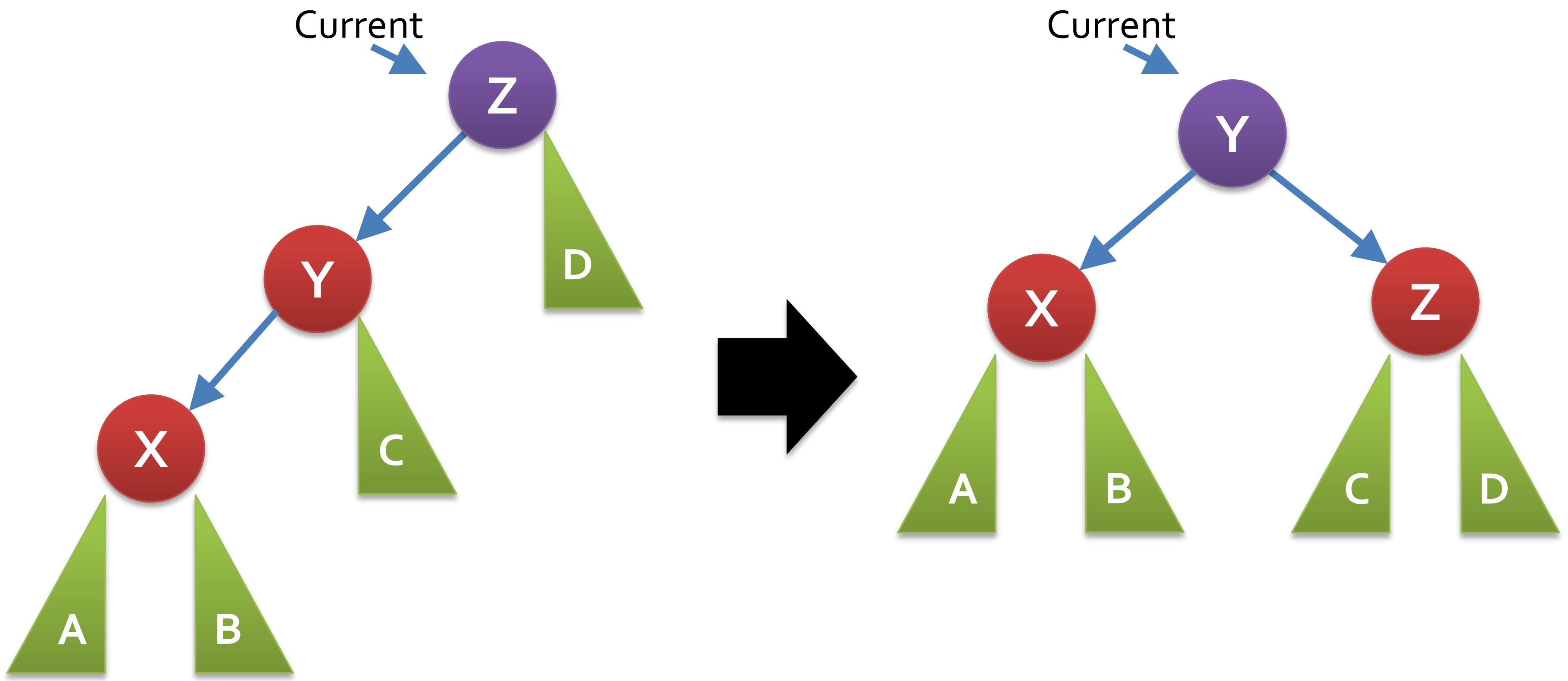
- We can do an insertion with a red-black tree using a series of rotations and recolors
- We do a regular BST insert
- Then, we work back up the tree as the recursion unwinds
  - If the right child is red and the left is black, we rotate the current node left
  - If the left child is red and the left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



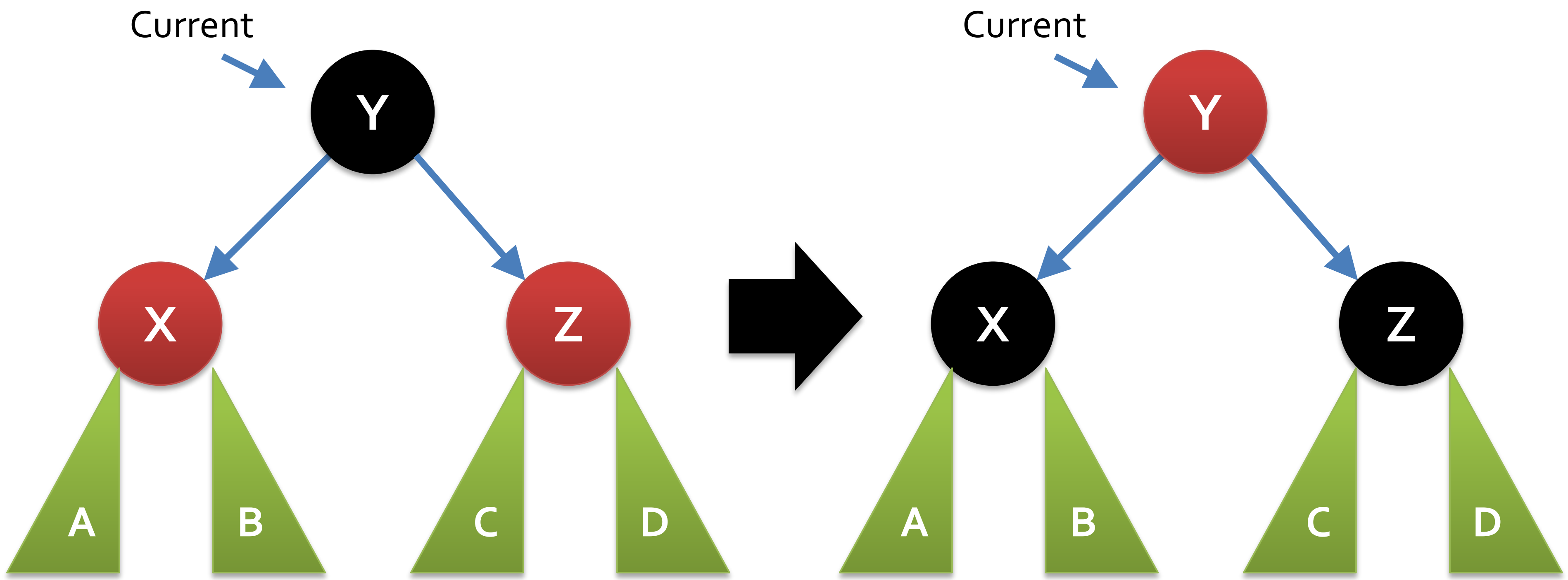
We perform a left rotation
when the right child is red

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-back 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***:
  - ***S*** has the starting node in it
  - ***V*** has everything else

1. Set the distance to all nodes in ***V*** to ∞
2. Find the node ***u*** in ***V*** with the smallest ***d***(***u***)
3. 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***)
4. Move ***u*** from ***V*** to ***S***
5. 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

Ticket Out the Door

Upcoming

Next time…

- Review everything after Exam 2
- More graph stuff
  - Eulerian tours and paths
  - NP-completeness
  - Matching
- B-trees
- Network flow
- Sorting
- Heaps
- Tries
- Review Chapters 2, 4, 5, and 6

Reminders

- Bring a question to class Wednesday!
  - Any question about any material in the course
- **Fill out course evaluations!**
- **Finish Project 4**
  - **Due Wednesday!**
- **Study for final exam**
  - **Friday, 12/05/2025 from 10:15 a.m. - 12:15 p.m.**