Java 2 Binary Search Trees

- A binary search tree is a **Binary Tree**
  - All of the things before apply to this
  - In which the **search property** holds on *every* node. A **Total Ordering**
  - That means duplicates would be on the same level, but on different sides of the node.
- A binary search tree imposes a **total order** on all its elements
- To write an ascending order iterator you would use an inorder traversal pattern
- Adding nodes can only go in the empty spots
- As you go toward the center the constraints on what can be added to the tree become more restrictive.

Searching for Values

- Begin at the root
- Use the search property (total order) of the nodes to guide the search through the tree.
- The amount of work to find a node is equal to the depth
  - The worst case is having to go the lowest node (which is equal to the height of the tree)
  - Searching a BST is O(height)
    - Unless it is balanced then it is O(log N)

- Use the search algorithm to locate the physical insertion point (Always adding Exactly one).
  - It only ever adds a new leaf node
  - Meaning worst case is the height of the tree O(height)
- A tree with a somewhat random insertion order is more favorable toward a balanced or complete tree
  - All trees are unique based on the order of insertion
- A tree with that adds in ascending order would produce the worst possible tree

Deleting Values
- Node to delete could be anywhere, not just a leaf.
  - The number of children that the node has determines how the value gets deleted from the tree.
  - Like if it has no children, its easy to delete, but if it has one or two children then your approach is different (Hubbard Deletion)
- The worst case deletion would be the lowest leaf, which is O(height)
- If its empty:
  - set the parent's pointer to null.
- If there is one child:
  - set the parent's pointer to the child os the deleted node.
- If there are two children:
  - Don't delete the node. Instead find a **replacement** for this node's vale and delete the node containing the replacement.
  - The replacement value is based off the inorder predecessor or successor of the node being replaced.

Random Adds
- If values are added in random order, the trees should stay relatively flat.
- Even more fun for COMP 3270: Expected number of compares for remove sort(N)

Balance
- A tree is balanced if a given lead is not much farther away from the root than any other leaf in a tree.
- A tree is balanced if for any given node, the left and right subtrees of that node have similar heights.
- Trees can be balanced **periodically** or **incrementally**
  - Periodically means we check it every so often then balance it.
  - Incrementally means we balance the tree before every add or remove.

Self-balancing search trees
- There are many different self-balancing search tree
- All SBSTs guarantee that the tree's height is O(log N) in the worst case, and that searching, inserting,

and deleting have worst case time comp O(Log N).

- AVL Trees
    - Very intuitive, but strict, and not as easy to use in code
- Red-Black Trees
    - Not as intuitive, but far more straightforward than AVL
    - Also a special type similar to red-black trees
- 2-4 Trees
    - Allows more than two children per node.
    - D - tree (basis of database systems)