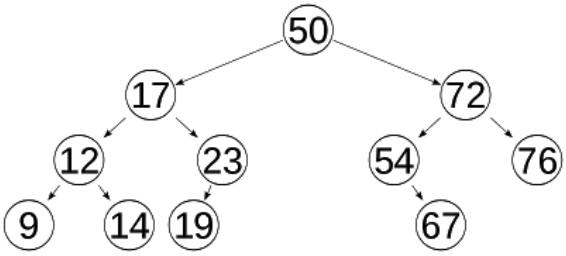
My BST (Binary search tree)

A ***Binary Search Tree*** (BST) is a specialized type of binary tree. It follows the same format as a "regular" binary tree (each node will be `null` or it will have some data member(s) and two references, `left` and `right`, to its child tree(s)). However, a BST is an "ordered" binary tree, where all the nodes are ordered (dynamically, as they're added).

For each node, all elements in its **left** sub-tree are **less** than the node, and all the elements in its right sub-tree are greater than the node (we're ignoring duplicates for now). Recursively, each of the sub-trees must also obey the same rules. Example:



Every node in the example above follows the constraints of a BST. Note that a BST is *not* necessarily "sorted" – it simply must obey the rules for the values of the sub-trees.

BST's have generally fast (O(log n)) lookup (as well as insertion and deletion) for any type of Comparable data. We will use integers as the data elements for the time being, for ease of comparison.

> *If you'd like to use generics (I suggest NOT using generics at first), reference types can't be compared with the < and > operators; the types stored in a generic BST must be Comparable objects. You accomplish this by declaring your MyBST class:* `class MyBST<T extends Comparable<T>>`*, saying it can store any type* `T` *as long as* `T` *is-a Comparable.*

Your MyBST class should have the following (in addition to necessary getters / constructors):

Instance variables `BSTNode root` – a reference to the overall root node in the tree

Inner class

```
private class BSTNode
{
	Integer val;
	BSTNode left, right;

	public BSTNode(Integer val) {
		this.val = val;
		left     = right = null;
	}

	@Override
	public String toString() { return "" + this.val; }
}
```

Methods

`int size()`	Return the number of elements in the tree
`void insert(Integer n)`	Add a node to the tree with value `n`
`boolean contains(Integer n)`	Find a node the tree with value `n`
`Integer getMax()`	Returns the largest value in the tree, or `null` if tree is empty
`Integer getMin()`	Returns the smallest value
`void delete(Integer n)`	Delete a node in the tree with value `n`. Does nothing if `n` doesn't exist in the tree
`void inOrder()`	Prints the contents of the tree by performing an "in-order" traversal. Does NOT print in "tree form" (this is handled by the `print` method)
`void print()`	Prints the state of the tree in a "tree like" format

In addition to the information on Canvas, some notes on individual methods:

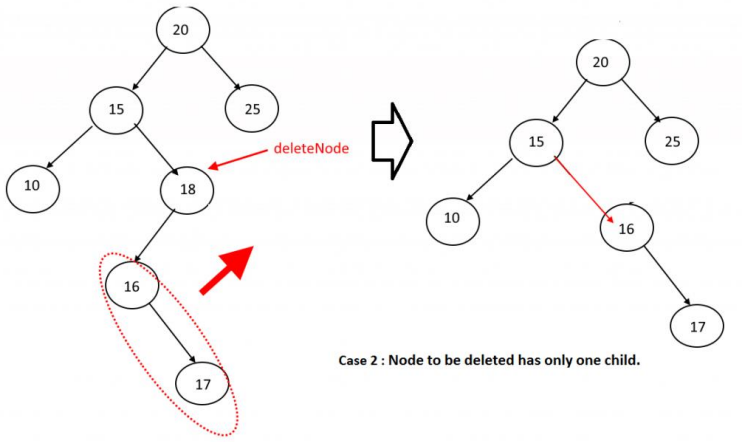
- `contains(Integer n)` - traverse the tree, (depending on the value of `n`) going left or right until the corresponding value is found, or until all the leaf nodes have been explored

- `insert(Integer n)` - similar to `contains`, but adds new nodes at their proper positions. (Hint: this could call a helper method that takes in 2 parameters `(BSTNode node, Integer n)`). Attempt to do this on your own; if you get stuck, algorithm help can be found on Canvas.

- `delete(Integer n)` - remove the supplied element from the tree (if it exists). Use a helper method. There are three special cases you need to consider, one of which is complicated:

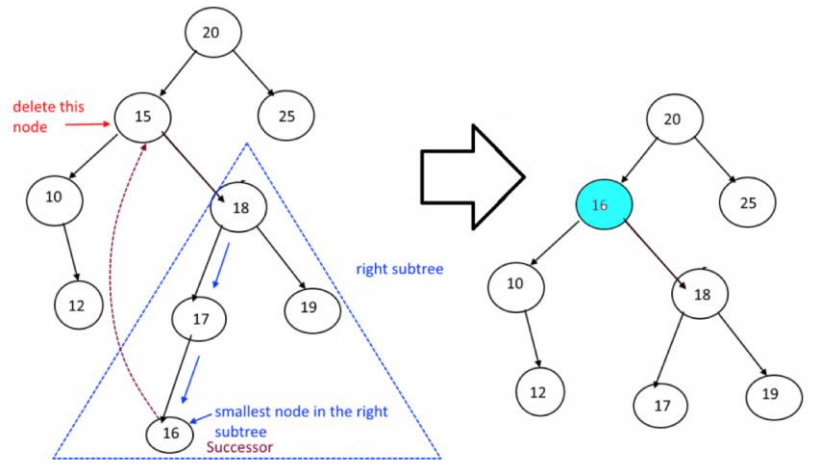
    - *Node to be deleted is a leaf* - the simplest case, just set the parent's reference to the node to be deleted to `null`.

    - *Node to be deleted has one child* - traverse to the node in question, keeping track of the parent node and the side on which the node exists. Find out which side is `null` and move the entire sub-tree (the reference to it) on the opposite side to the parent on the side the node-to-be-deleted existed ("link past" the node to be deleted). Example:



- *Node to be deleted has two children* - a more complex situation. To perform this operation, you must find the "successor" node. The successor is the node which will replace the deleted node. The successor is the smaller node in the right sub-tree of the node to be deleted.

If you get stuck on the `delete` method help can be found on Canvas.



- `print()` - Output the state of the tree in (horizontal) tree-like form. The text for a node should be indented four times the depth of the node (for example a node at level 0 (the overall root) would not be indented; a node at depth 3 would be indented 12 spaces).

  For a tree containing the values [5, 2, 6, 1, 3, 9] (inserted in that order) the output should be:

```
        9
    6
5
        3
    2
        1
```

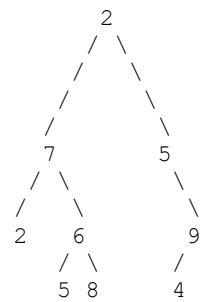
When you're done, use the Runner class (provided, in the lab folder) to test your code. Your output should match that in the **"output.txt"** file, in the same location.

(Advanced) Create a nicer printing tree

Add a method to make your tree class printable in a more aesthetically pleasing fashion. You could again print the tree "on its side", or print a tree that grows "down":

```
                /----- 20
        /----- 14
        |       \----- 13
/----- 12
|       |       /----- 11
|       \----- 10
|               \----- 9
8
|               /----- 7
|       /----- 6
|       |       \----- 5
\----- 4
        |       /----- 3
        \----- 2

                \----- 1
```



(Over 9000) The Great Tree-List Recursion Problem

Solve the problem described as "one of the neatest recursive pointer problems ever devised" by Nick Parlante, creator of codingbat.com and computer science professor at Stanford. This problem "uses pointers, binary trees, linked lists, and some significant recursion."

You can find the problem description here: cslibrary.stanford.edu/109/TreeListRecursion.html