Notes on Binary Search Trees (BST)

Definition:

For each internal node in a BST, the key of that node is greater than the key of all nodes in the left subtree, and less than the key of all nodes in the right subtree.

Equivalently, all keys in the left subtree of a node must be less than the key of that node, and all keys in the right subtree of a node must be greater than the key of that node.

Traversals:

In each of these snippets, visit means doing the work that we need to do at each node ("processing" it).

Preorder:

```
preorder(Node node):
    if node != null:
        visit(node)
        preorder(node.left)
        preorder(node.right)
```

Inorder:

```
inorder(Node node):
    if node != null:
        inorder (node.left)
        visit(node)
        inorder(node.right)
```

Postorder:

```
postorder (Node node):
    if node != null:
        postorder(node.left)
        postorder (node.right)
        visit(node)
```

Searching for a node: (contains/find)

```
search(Node node, int item):   // we use ints as an example, but can be anything
    if node == null:
        return false                        // not found
    else if item == node.key:
        return true                         // found
    else if item <  node.key:
        return search(node.left, item)      // search left subtree
    else // item >  node.key:
        return search(node.right, item)     // search right subtree
```

Start the algorithm above with `search(root, item).`

Adding a new node

```
add(Node node, int newKey):
    if newKey == node.key:
        return;                             // already in tree
    else if newKey < node.key:
        if node.left == null:
            add new node at node.left with the new key
        else:
            add(node.left, newKey)
    else // newKey > node.key:
        if node.right == null:
            add new node at node.right with the new key
        else:
            add(node.right, newKey)
```

Deletion

- Find (using the search/contains algorithm) the node you want to delete.
- 3 cases:
    - If the node has no children, just delete it (remove it from the tree).
    - If the node has one child, delete the node and move the node's single child into the same location where the original node was.
    - If the node has two children, replace the *value* in the node to be deleted with the *value* of the node's inorder successor.